

Reconfigurable Computing Platforms and Target System Architectures for Automatic HW/SW Compilation

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des Grades Doktor-Ingenieur (Dr.-Ing.)

von

Dipl.-Inform. Holger Lange

geboren in Lehrte

Referenten: Prof. Dr. Andreas Koch
Prof. Dr. Ulrich Golze

Datum der Einreichung: 23.02.2011
Datum der mündlichen Prüfung: 11.04.2011

Darmstadt, 2011
Hochschulkennziffer: D 17

For Daniela.

Abstract

Embedded systems found their way into all areas of technology and everyday life, from transport systems, facility management, health care, to hand-held computers and cell phones as well as television sets and electric cookers. Modern fabrication techniques enable the integration of such complex sophisticated systems on a single chip (System-on-Chip, SoC). In many cases, a high processing power is required at predetermined, often limited energy budgets. To adjust the processing power even more specifically to application needs while meeting the energy budget, Field-Programmable Gate Arrays (FPGAs) can be employed as energy-conserving configurable logic devices. Beyond their well-known flexibility, recent FPGAs offer sufficient capacity to accommodate entire SoCs. These reconfigurable SoCs (rSoCs) can serve as a basis for Reconfigurable Computing Platforms (RCPs). Here, a suitable part of the application program is compiled into hardware (HW) to increase efficiency, while the remainder is executed in software (SW). Beyond the logic circuits themselves, RCPs require an infrastructure for platform and application design, program execution, and application data storage to accommodate the increasing complexity of modern embedded systems.

Hence, this work defines four Columns supporting the RCP. As the first Column, this thesis presents an execution model orchestrating the fine-grained interaction of a conventional general purpose processor (GPP) and a high-speed reconfigurable hardware accelerator (HA), the latter having full direct access to memory. The resulting requirements are realized efficiently in a custom computer architecture by a number of solutions on both the hardware as well as operating system levels. One of these measures is a low-latency HA-to-GPP signalling scheme that reduces the response time by up to a factor of 23x even on a relatively slow embedded processor. Another one is a high-bandwidth shared memory interface that does not interfere with time-critical operating system functions executing on the GPP, but still makes 89% of the theoretical memory bandwidth available to the HA. Two schemes of allowing the HA access to protected virtual memory complement the physical interface, differing in their use of an MMU, and their flexibility / performance trade-offs. All of the techniques and their interactions are then evaluated at the system level using the full-scale virtual memory variant of the Linux operating system.

The second Column is formed by the description of static RCP characteristics. Complex embedded systems often comprise large system-specific parameter sets that span a wide range of possible combinations, not all of them legal. On the one hand, the RCP description is important as a well-defined target for EDA tools and compilers. On the other hand, intellectual property (IP) cores targeted at (r)SoCs expect a specific system environment, certain resources, or address windows to map their registers. Hence, an RCP management system is presented for describing static component properties,

interfaces, and bus systems. To this end, the RCP management not only considers individual components and their configuration parameters, but presents a homogeneous view of the entire system. However, the static RCP description does not suffice to serve as a compiler target. In addition, a powerful RCP resource and component management is defined, which enables compile-time reservation of resources requested by IP cores. Moreover, an algorithm for automatic RCP composition is specified, which relies on the RCP description and interacts closely with the resource management to instantiate RCP components and connect them via bus systems.

A description of dynamic RCP characteristics is further required to embed complex IP cores from high-level languages into automatically generated datapaths, and forms the third Column supporting the RCP. The resulting multitude of possible interface combinations requires the exploration of a large design space when manually developing the system. To compose IP cores tightly coupled with the datapath from an ANSI C description into a system, and subsequently interact with them, a HW/SW interface description was developed. It establishes an automatic design flow presenting convenient, simple C interfaces (function prototypes) to a SW developer. The description consists of rules for an idiomatic C programming style, and the interface control semantics of an IP core. Correspondingly, it defines a data model and human-readable description language for the characteristics of individual IP cores. The interplay of these components is evaluated on an rSoC using a real-world example.

Sourcing and draining the data volume of a large embedded system without latency while intelligently exploiting the limited memory bandwidth can be challenging. Hence, a distributed speculative memory system is presented as the fourth Column supporting the RCP. By exploiting the reconfigurability of the RCP, the highly parameterized memory system can be adapted to the specific needs of each application. Customizable characteristics include the number of parallel memory ports to support the spatially distributed computation model of HAs. In contrast to the temporally distributed one of an SPP, each memory operator can be connected to a dedicated port here. Efficient speculative execution of the HA datapath is enabled by a dynamic prioritization scheme for arbitrating access to shared memory. The prioritization depends on per-port control speculation statistics collected at run-time, preferably serving accesses with a higher execution probability. This scheme, inspired by the branch prediction of conventional GPPs, additionally employs speculative prefetching to make very probably required data available in the cache in time before the actual access. System-level measurements on an rSoC show speed-ups of up to 1.65x at only low resource usage for typical applications.

Kurzfassung

Eingebettete Systeme haben Einzug in alle Bereiche der Technik und des täglichen Lebens von Verkehrssystemen, Gebäudemanagement, Medizin, über Taschencomputer und Mobiltelefone bis hin zu Fernsehgeräten und Elektroherden gehalten. Moderne Fertigungsmethoden erlauben die Integration solcher komplexen, hochentwickelten Systeme auf einem einzigen Baustein (System On Chip, SoC). Um die oft hohe benötigte Rechenleistung unter Einhaltung des vorgegebenen, zum Teil eingeschränkten Energiebudgets noch spezieller auf den Anwendungsfall abzustimmen, können Field-Programmable Gate Arrays (FPGAs) als energiesparende, programmierbare Logikbausteine eingesetzt werden. FPGAs bieten neben ihrer bekannten Flexibilität mittlerweile ausreichend Kapazität, um gesamte SoCs aufzunehmen. Diese rekonfigurierbaren SoCs (rSoCs) können als Basis für Rekonfigurierbare Rechenplattformen (RR) dienen. Bei RRs wird zur Effizienzsteigerung ein geeigneter Teil des Anwendungsprogramms in Hardware (HW) übersetzt, während der Rest in Software (SW) ausgeführt wird. Zur Beherrschung der steigenden Komplexität moderner eingebetteter Systeme benötigen RRs über die reine Schaltungslogik hinaus eine Infrastruktur zur Plattform- und Anwendungserstellung, Programmausführung und Speicherung von Anwendungsdaten.

Diese Arbeit definiert daher vier Säulen zur Unterstützung der RR. Als erste Säule wird ein Ausführungsmodell zur Steuerung der feingranularen Interaktion zwischen einem konventionellen SW-programmierbaren Prozessor (SPP) und einem schnellen rekonfigurierbaren HW-Beschleuniger (HB) vorgestellt, wobei der HB über vollständigen direkten Speicherzugriff verfügt. Anschließend wird die effiziente Umsetzung der resultierenden Anforderungen durch eine Reihe von Lösungen auf HW- und Betriebssystem (BS)-Ebene in einer maßgeschneiderten Rechnerarchitektur beschrieben. Eine dieser Maßnahmen ist ein Signalisierungssystem zwischen HB und SPP mit niedriger Latenz, welches die Antwortzeit sogar auf einem relativ langsamen eingebetteten Prozessor bis zu 23-fach reduziert. Eine weitere Lösung bietet eine verbesserte Schnittstelle mit hohem Datendurchsatz für Zugriffe auf gemeinsam genutzten Speicher. Dabei werden auf dem SPP ausgeführte, zeitkritische BS-Funktionen nicht beeinträchtigt, obwohl der HB gleichzeitig 89% der theoretischen Speicherbandbreite nutzen kann. Ergänzend zur physikalischen Schnittstelle werden zwei Schemata vorgestellt, die dem HB den Zugriff auf geschützten virtuellen Speicher ermöglichen. Diese unterscheiden sich in der Verwendung einer Speicherverwaltungseinheit (MMU) sowie gegenläufig ausgeprägter Flexibilität und Leistung. Alle Techniken und ihr Zusammenspiel werden dann auf Systemebene unter Einsatz der vollständigen, virtuellen Speicher nutzenden Variante des Linux-BS evaluiert.

Die Beschreibung der statischen Eigenschaften von RR bildet die zweite Säule. Komplexe eingebettete Systeme umfassen oft große systemspezifische Parametermengen, die einen weiten Bereich möglicher, nicht immer zulässiger Kombinationen überdecken.

Der RR-Beschreibung kommt einerseits eine besondere Bedeutung als wohldefiniertes Ziel für automatische Entwurfswerkzeuge und Compiler zu, andererseits erwarten auf (r)SoC ausgerichtete Intellectual Property (IP)-Blöcke eine bestimmte Systemumgebung, spezielle Ressourcen oder Adressbereiche zur Einblendung ihrer Register. Daher wird ein RR-Verwaltungssystem zur Beschreibung statischer Komponenteneigenschaften, -schnittstellen und Bussysteme vorgestellt. Dabei werden nicht nur einzelne Komponenten und deren Konfigurationsparameter abgebildet, sondern es wird eine einheitliche Sicht auf das Gesamtsystem bereitgestellt. Um als Compiler-Zielplattform dienen zu können, wird über die statische RR-Beschreibung hinaus ein leistungsfähiges Management der RR-Ressourcen und -komponenten definiert, welches die Reservierung für IP-Blöcke benötigter Ressourcen zur Übersetzungszeit ermöglicht. Ferner wird ein Algorithmus zur automatischen RR-Komposition (Instanzierung der einzelnen RR-Komponenten und deren Verbindung durch Bussysteme) ausgehend von der RR-Beschreibung angegeben, welcher eng verzahnt mit dem Ressourcenmanagement arbeitet.

Eine Beschreibung von dynamischen RR-Eigenschaften wird weiterhin für die Einbindung von komplexen IP-Blöcken aus Hochsprachen in automatisch generierte Datenpfade benötigt, der dritten RR-tragenden Säule. Die dabei entstehende Vielzahl möglicher Schnittstellenkombinationen erzwingt bei manueller Systementwicklung die Erforschung eines großen Entwurfsraums. Um IP-Blöcke ausgehend von einer ANSI C Beschreibung eng gekoppelt mit dem Datenpfad zu einem System zusammenzusetzen und mit ihnen zu interagieren, wurde eine HW/SW Schnittstellenbeschreibung entwickelt. Sie bietet einen automatischen Design-Fluss, wobei sie einem SW-Entwickler geeignete, einfache C Schnittstellen (Funktionsprototypen) präsentiert. Sie besteht aus Regeln für einen idiomatischen C Programmierstil und den Schnittstellensteuerungssemantiken eines IP-Blocks. Dazu definiert sie ein Datenmodell und eine menschenlesbare Beschreibungssprache für die Charakteristiken einzelner IP-Blöcke. Das Zusammenspiel dieser Komponenten wird anhand eines realen Beispiels auf einem rSoC evaluiert.

Um das Datenaufkommen eines großen eingebetteten Systems verzögerungsfrei zu verarbeiten und dabei die begrenzte verfügbare Speicherbandbreite intelligent zu nutzen, wird als vierte RR-tragende Säule ein verteiltes spekulatives Speichersystem präsentiert. Dazu kann das durchgehend parametrisierte Speichersystem durch Ausnutzung der Rekonfigurierbarkeit der RR an die Erfordernisse der jeweiligen Anwendung angepasst werden. Unter anderem ist die Anzahl der parallelen Speicherports wählbar, um das räumlich verteilte Berechnungsmodell der HBs zu unterstützen. Im Gegensatz zum zeitlich verteilten Modell der SPPs kann hier jeder Speicheroperator an einen eigenen Port angeschlossen werden. Dabei wird die effiziente spekulative Ausführung des HB-Datenpfads durch dynamisch priorisierte Zugriffe der einzelnen Ports auf den gemeinsam genutzten Speicher ermöglicht. Die Prioritätsvergabe richtet sich nach portweise zur Laufzeit erhobenen Kontrollspekulationsstatistiken, wobei Zugriffe mit höherer Ausführungswahrscheinlichkeit bevorzugt bedient werden. Dieses von der Sprungvorhersage konventioneller SPPs inspirierte Verfahren verwendet darüber hinaus spekulatives Prefetching, um sehr wahrscheinlich benötigte Daten rechtzeitig vor dem tatsächlichen Zugriff im Cache bereitzustellen. Messungen auf Systemebene an einem rSoC zeigen eine bis zu 1,65-fache Beschleunigung typischer Anwendungen bei nur geringem Ressourcenverbrauch.

Acknowledgements

First of all, I would like to thank my thesis advisor Prof. Andreas Koch for many inspiring discussions that prompted me to explore and develop solutions for reconfigurable computing platforms. Andreas has been known to me since my first steps into scientific work and introduced me to a wide array of challenges in the embedded systems domain. Besides his valuable input, he also provided the technical equipment essential for the success of my work.

I am also thankful to Prof. Ulrich Golze for accepting to be a co-referee for this thesis. Prof. Golze taught me the foundations of hardware design during my studies, and thus kindled my interest in custom computer architecture.

Furthermore, I am grateful to my present and past colleagues in Darmstadt and Braunschweig for many fruitful discussions and feedback on my research ideas. Among them, I am particularly indebted to Florian, who helped identifying pitfalls by tirelessly scrutinizing potential approaches.

Free access to open source software was instrumental in developing this work, especially the Linux kernel and userland tools. I owe many thanks to all volunteers in the open source community.

I dearly thank my parents who laid the foundations and gave me the freedom and support that enabled me to successfully accomplish this endeavor.

Finally, I saved my very special thanks for my loving wife Daniela, who supported me morally during the ups and downs of this project and patiently bore my corresponding moods. Beyond that, she offered invaluable advice on the illustrations, and many comments that improved the legibility of the text.

“I don’t hold with those newfangled things.”

John Steed

The Avengers, *“The Cybernauts”*, 1965

This work was partly funded by DFG and LOEWE AdRIA.

Contents

1. Introduction	1
1.1. System Architecture	3
1.2. Tool Support	5
1.3. Execution Model	6
1.4. Platform Management	7
1.5. HW/SW Interface	8
1.6. Speculative Memory System	8
1.7. Reconfigurable Computing Platform	9
1.8. Thesis Contributions	9
1.9. Thesis Structure	11
2. Overview of Reconfigurable Computing Platforms	13
2.1. Data Exchange Modes	14
2.2. RCP System Architecture	15
2.2.1. Standalone	15
2.2.2. System Bus	16
2.2.3. Processor Local Bus	17
2.2.4. Processor Pipeline	19
3. Prior and Related Work	21
3.1. First Column: Execution Model	21
3.1.1. Architectures Combining GPPs with HAs	22
3.1.2. Virtually-addressed Shared Memory	22
3.1.3. Simulated Environments	24
3.2. Second Column: Platform Management	24
3.2.1. Configuration Management	24
3.2.2. Platform Libraries	25
3.2.3. Platform-Based Design	26
3.2.4. Parameterized Applications	26
3.2.5. Modeling Languages and Configuration Management	27
3.3. Third Column: HW/SW Interface	28
3.3.1. Architecture Description Languages	28
3.3.2. HW/SW Compilers	28
3.3.3. Interface Synthesis	30
3.4. Fourth Column: Speculative Memory System	30
3.4.1. Hardware-based Memory Speculation	30

3.4.2.	Data Prefetching	31
3.4.3.	Hardware Transactional Memory	32
4.	Execution Model	37
4.1.	Execution Models for Hardware Acceleration	38
4.2.	Platform Requirements	41
4.3.	Target Platform	42
4.3.1.	Vendor Flow for rSoC Composition	44
4.3.2.	Practical Limitations	46
4.4.	OS Integration: Low-Latency GPP↔HA Communication	47
4.5.	High-Performance HA Memory Access	49
4.5.1.	Abstracting Memory Interfaces	51
4.5.2.	Cooperative System Architecture	52
4.6.	OS Integration: Virtual Memory	53
4.6.1.	Initial Approach: In-Memory DMA Buffer	53
4.6.2.	Refined Solution: AISLE	55
4.6.3.	Full Virtual Memory Support in the HA: PHASE/V	59
4.7.	Chapter Summary	61
5.	Platform Management	63
5.1.	Data Model	64
5.2.	Notation and Basic Grammar	66
5.3.	Parameters	69
5.4.	Interfaces	74
5.4.1.	Interface Binding	75
5.4.2.	Ports	79
5.5.	Components	98
5.5.1.	Dependency Relations	102
5.5.2.	Examples for Dependency Relations	103
5.6.	Platforms	104
5.7.	CoMAP Tools	107
5.8.	Real World Example: Xilinx ML507 rSoC	115
5.9.	Chapter Summary	116
6.	Interfacing C Language Software and IP Core Hardware	117
6.1.	Problem Description	118
6.2.	System Model	119
6.3.	Hardware Interface	123
6.3.1.	Static Interface Properties	124
6.3.2.	Dynamic Interface Properties	125
6.4.	Software Interface	129
6.4.1.	Primitives	130
6.4.2.	Monoliths	131
6.4.3.	Mapping C Function Parameters	132

6.5.	Compiler Integration	133
6.5.1.	Comrade Compiler System	134
6.5.2.	PaCIFIC Extension to Comrade	136
6.5.3.	Real World Example: Automatic FFT IP Core Integration	137
6.6.	Chapter Summary	143
7.	Speculative Memory System	145
7.1.	Speculative Program Execution	146
7.1.1.	Speculation Types	146
7.1.2.	Memory-based Speculation	149
7.1.3.	Memory Speculation: GPP vs. ACS Execution Models	153
7.2.	Memory System Requirements	158
7.2.1.	Lightweight Non-destructive Speculation	158
7.2.2.	Speculating Memory Writes: Load Store Queue	158
7.2.3.	Discussion	162
7.3.	Efficient Implementation	163
7.3.1.	Target Platform	163
7.3.2.	Lightweight Control Speculation	164
7.4.	System Integration: FastLane+ and MARC	169
7.4.1.	User Interface Signals	172
7.5.	System Integration: CoMAP	173
7.6.	Chapter Summary	175
8.	Experimental Results	177
8.1.	Execution Model	177
8.1.1.	FastPath	177
8.1.2.	FastLane+	179
8.1.3.	AISLE versus PHASE/V	182
8.2.	CoMAP and PaCIFIC	183
8.3.	Speculative Memory System	187
8.3.1.	Linked List	187
8.3.2.	Tree Search	190
8.3.3.	Merge Sort	192
8.3.4.	Comrade Sample	195
8.3.5.	Implementation Aspects	196
8.4.	Power and Energy Consumption	197
9.	Summary and Future Work	199
9.1.	Lessons Learned	202
9.2.	Future Research	203
	Bibliography	205
A.	Appendix	223

List of Figures

1.1. The four Columns of a Reconfigurable Computing Platform	2
1.2. Basic FPGA fabric architecture	4
1.3. A reconfigurable system-on-chip on a Xilinx FPGA (adapted from [Xili09a])	5
1.4. Reconfigurable Computing Platform overview diagram	10
2.1. Slave and master mode communication between GPP, RD, and memory	14
2.2. Stand-alone RD-PE attached via CAN peripheral interface	15
2.3. ACE-V adaptive computer with RD-PE attached via i960 system bus	16
2.4. Xilinx ML507 reconfigurable SoC with RD-PE attached via processor local bus	17
2.5. Stretch SCP with RD-PE attached at Execute stage of processor pipeline	19
4.1. ASH execution model	38
4.2. Sample program with HA-unsuitable statements	39
4.3. Example in the Nimble execution model	39
4.4. Example in the Comrade execution model	40
4.5. Xilinx ML310 with Virtex-II Pro (adapted from Xilinx manuals)	42
4.6. ML310 system (from Xilinx manuals)	43
4.7. HA integration via OPB	44
4.8. HA integration via PLB	45
4.9. FastPath: Low-latency SW calls and live variable transfers	48
4.10. Fast Variable Exchange	49
4.11. FastLane+: Attaching HA directly to DDR controller	50
4.12. MARC overview	51
4.13. FastLane+ with MARC interface	52
4.14. HW and SW addressing of memory	54
4.15. API example for DMA Buffer	55
4.16. Conventional and HA-compatible program layouts	56
4.17. API example for AISLE	58
4.18. PHASE/V TLB system: HA↔OS interactions	60
4.19. PHASE/V TLB system: Tag- and translation RAM	61
5.1. CoMAP data model	64
5.2. Simplified Xilinx ML507 platform represented by CoMAP elements	65
5.3. Hierarchical parameters in VHDL [LaRa02]	70
5.4. CoMAP parameter definition	73
5.5. CoMAP interface definition using a template	75

5.6.	Interface binding process	77
5.7.	CoMAP interface binding	78
5.8.	CoMAP clock port	81
5.9.	CoMAP reset port	82
5.10.	Linking a data port to an address port	83
5.11.	Physical and logical address representations	83
5.12.	Address translation using ranges	84
5.13.	A bus bridge translates address ranges	84
5.14.	Address ranges are translated via an internal representation	85
5.15.	A sequence defines the data protocol for an abstract data type	86
5.16.	Offsets measured between handshakes and port data	89
5.17.	Latencies measured between two handshakes	89
5.18.	Handshakes with interrupt and data semantics	90
5.19.	A read with acknowledge triggered by an interrupt	90
5.20.	Interface logic implementing the described handshake protocol	91
5.21.	Bus arbitration between four master-capable ports in two priority classes	93
5.22.	Bus arbiter: Four ports in two priority classes	93
5.23.	Gaussian traffic distribution	96
5.24.	Example for gaussian traffic scheme (also shown in Figure 5.23)	96
5.25.	A local traffic scheme	97
5.26.	Discrete traffic distribution	97
5.27.	A discrete traffic scheme	97
5.28.	A linear traffic scheme	98
5.29.	Components derived as instances from a master IP core	99
5.30.	A simple component definition	101
5.31.	Platform composition using active components and interfaces templates	105
5.32.	A platform definition	107
5.33.	Sample input files for CoMAP-VPP	109
5.34.	Output files generated by CoMAP-VPP from input shown in Figure 5.33	110
5.35.	CoMAP-VPP user interface	111
5.36.	Sample input files for Parameter Extractor	112
5.37.	CoMAP repository entry for extracted free parameter	113
5.38.	Finding a valid address mapping	114
5.39.	CoMAP address mappings for both bus bridges shown in Figure 5.38	114
5.40.	Subset of Xilinx ML507 rSoC reference design	115
6.1.	Hardware pipeline used by software	118
6.2.	IP cores as C functions	118
6.3.	PaCIFIC system model and design flow	120
6.4.	Part of CoMAP interface for the crypt IP	124
6.5.	UCODE pipelining model (from [Koch07])	128
6.6.	UCODE pipeline administration logic for an IP core with pipeline depth 4, PaCIFIC extension is a repeat counter (based on [Koch07])	129
6.7.	UCODE for behavior “encrypt”	131

6.8. Signal timing for the crypt IP core	132
6.9. CoMAP sequence definition for port <i>INDATA</i>	132
6.10. Comrade compiler flow	134
6.11. Control flow before and after split	136
6.12. Pipeline of combined HW nodes	137
6.13. FFT pipeline controlled by PaCIFIC	137
6.14. C program calling FFT hardware accelerator	138
6.15. Intermediate representation (IR) of FFT application	138
6.16. UCODE for behavior “fft16”	139
6.17. CoMAP description for ports DI and XK	140
6.18. IR node containing basic block before and after split	141
6.19. Stream engines operating from pointers source and sink memory areas . .	142
7.1. Program flow with control speculation	147
7.2. Program flow with data speculation	147
7.3. Program flow with combined control and data speculation	148
7.4. Read after write (RAW) hazard	150
7.5. Write-after-read (WAR) hazard	150
7.6. Write-after-write (WAW) hazard	150
7.7. Memory hierarchy for memory-based speculation	151
7.8. Speculative writes generate multiple versions of the same memory address	152
7.9. Memory updates are only visible in the same coherent control branch . .	153
7.10. A write buffer disambiguates write operations	154
7.11. C program fragment compiled into parallel datapath (Figure 7.12)	155
7.12. Parallel datapath - each memory operation uses a separate memory port	155
7.13. Parallel datapath - speculative execution of memory accesses	156
7.14. Parallel datapath - sequence numbers represent original program order .	157
7.15. Load Store Queue Data Structure	159
7.16. Distributed LSQ with parallel look-ups and serialized updates (writes) . .	161
7.17. MARC CachePort interface with alternating reads and writes	164
7.18. Speculative memory ports are a front end to MARC’s CachePorts	165
7.19. Two-level GAg predictor	167
7.20. Bitonic sorting network assigns dynamic priorities	167
7.21. A single CAM cell with fast write capability	169
7.22. New MARC front end for prioritizing control speculation	170
7.23. Original non-speculative and new speculative CachePort FSM	171
7.24. Cancelling and committing speculative reads	172
7.25. Cancelling and committing speculative writes	173
7.26. Cancelling non-speculative reads	173
7.27. CoMAP traffic patterns inducing static and dynamic memory port priorities	174
8.1. Effective speed-up as function of HA execution time and raw HW acceleration factor for different latencies	178

8.2. C code of the pointer chasing application	182
8.3. FFT pipeline controlled by PaCIFIC (duplicate of Figure 6.13 reproduced here for convenience)	184
8.4. Stream engines operating from pointers source and sink memory areas (duplicate of Figure 6.19 reproduced here for convenience)	185
8.5. C code of the Linked List application (16 bytes per element)	188
8.6. C code of the Tree Search application (16 bytes per element)	191
8.7. C code of the Merge Sort application [EEMB11]	193
8.8. C source code of the Comrade Sample application	195

List of Tables

4.1. Summary of platform requirements	42
4.2. PLB specification vs. implementation	46
4.3. Area overhead in cells (LUT+FF) for bus wrappers in vendor flow	47
5.1. Parameter classes in selected file formats	70
5.2. Parameter properties	70
5.3. Interface properties	75
5.4. Port properties	80
5.5. Sequence properties	85
5.6. Handshake properties	87
5.7. Handshake types	87
5.8. Comparison with common handshaking terms	88
5.9. Access properties	92
5.10. Traffic properties	94
5.11. Component properties	99
5.12. Example specification of technologies and their properties	102
5.13. Platform properties	105
5.14. CoMAP tools and categories	108
6.1. IP core interface classification	120
7.1. LSQ CAM area and timing for typical numbers of entries and memory ports	162
8.1. FPGA areas for rSoCs with specified HA at 100 MHz clock	180
8.2. Copy-HA runtimes and available throughput using V2P reference design and FastLane+ memory subsystem implementations under various GPP SW loads	180
8.3. Software run times and slow-down on idle system and using Copy-HA attached by V2P reference design and FastLane+ memory subsystem implementations	181
8.4. Runtimes of the pointer chasing application	183
8.5. FPGA areas for the FFT application with CoMAP/PaCIFIC	186
8.6. Runtimes of the FFT application with CoMAP/PaCIFIC	186
8.7. Runtimes and predictor accuracies of the Linked List application (random list layout, 50% writes)	188
8.8. Runtimes and predictor accuracies of the Linked List application (random list layout, 80% writes)	189

8.9. Runtimes and predictor accuracies of the Linked List application (random list layout, 20% writes)	189
8.10. Runtimes and predictor accuracies of the Linked List application (sequential list layout, 50% writes)	189
8.11. Runtimes and predictor accuracies of the Tree Search application (keys in insertion order)	191
8.12. Runtimes and predictor accuracies of the Tree Search application (keys in ascending order)	192
8.13. Runtimes and predictor accuracies of the Merge Sort application (cache flushes)	194
8.14. Runtimes and predictor accuracies of the Merge Sort application (shared cache)	194
8.15. Runtimes and predictor accuracies of the Merge Sort application (shared cache, without live variable transfer overhead)	194
8.16. Runtimes and predictor accuracies of the Comrade Sample application . .	196
8.17. FPGA areas for rSoCs with specified application and memory-based speculation	196
8.18. ML310 board-level supply currents and total power consumption for specified combinations of hardware accelerator and SW application	197
8.19. Total energy consumption for the pointer chasing application executed in SW and as HW accelerator	198

Listings

- A.1. CoMAP interface templates for Xilinx ML507 223
- A.2. CoMAP component library for Xilinx ML507 235
- A.3. CoMAP platform definition for Xilinx ML507 256

Abbreviations

ABI	Application Binary Interface	Comrade	Compiler for Adaptive Environments
ACS	Adaptive Computing System	CPU	Central Processing Unit
ADDR	Address	DDR	Double Data Rate
ADL	Architecture Description Language	DMA	Direct Memory Access
ADT	Abstract Data Type	DRAM	Dynamic Random Access Memory
AE	Application Engine	DSP	Digital Signal Processor
AISLE	Accelerator-Integrating Shared Layout for Executables	EDA	Electronic Design Automation
ALU	Arithmetic Logic Unit	EDK	Embedded Development Kit
AMBA	Advanced Microcontroller Bus Architecture	ELF	Executable and Linking Format
API	Application Programming Interface	ESL	Electronic System Level
ARB	Address Resolution Buffer	FIFO	First In, First Out
ATX	Advanced Technology Extended	FIR	Finite Impulse Response
BRAM	Block Random Access Memory	FPGA	Field-Programmable Gate Array
BTB	Branch Target Buffer	FSB	Front Side Bus
CAM	Content-Addressable Memory	FSM	Finite State Machine
CAN	Controller Area Network	GA_g	Global pattern history, Adaptive predictor, global predictor table
CFG	Control Flow Graph	GID	Group Identifier
CMDFG	Control Memory Data Flow Graph	GPP	General Purpose Processor
CoMAP	Configuration Manager for Abstract Parameterizations	HA	Hardware Accelerator
CoMAP-VPP	CoMAP-Verilog PreProcessor	HAMEM	High-throughput HA memory access
		HDL	Hardware Description Language

HDTV High-Definition Television	OP Output Prepare
HTM Hardware Transactional Memory	OPB On-chip Peripheral Bus
IEC International Electrotechnical Commission	OS Operating System
ILP Instruction-Level Parallelism	OSSCHED Hardware accelerator must obey OS scheduler
IP Intellectual Property	PaCIFIC Parametric C Interface for IP Cores
IR Intermediate Representation	PCB Printed Circuit Board
IRQ Interrupt Request	PCI Peripheral Component Interconnect
LOWLAT Low-latency GPP↔HA communication	PCI-Express Peripheral Component Interconnect Express
LSQ Load Store Queue	PCI-X Peripheral Component Interconnect eXtended
LUT Look-Up Table	PDA Personal Digital Assistant
MAC Media Access Control	PE Processing Element
MARC Memory Architecture for Reconfigurable Computers	PHT Pattern History Table
MARC II MARC version II	PHY Physical layer of the Open Systems Interconnection model
MARTE Modeling and Analysis of Real-Time and Embedded Systems	PLB Processor Local Bus
MCI Memory Controller Interface	PROTCODE SW code protected from hardware accelerator access
ML310 Xilinx ML310 embedded development platform	PROTSYS Hardware accelerator access confined to own process
ML507 Xilinx ML507 embedded development platform	QDR Quad Data Rate
MMU Memory Management Unit	QoS Quality of Service
MUXCY Carry chain Multiplexer	QPI Quick Path Interconnect
NRE Non-Recurring Engineering	RAM Random Access Memory
NUMA Non-Uniform Memory Architecture	RAW Read After Write
OE Output Enable	RCP Reconfigurable Computing Platform

RCU Reconfigurable Compute Unit	SWPERF Hardware accelerator may not slow down software
RD Reconfigurable Device	
RFID Radio-Frequency Identification	TID Transaction Identifier
rSoC reconfigurable System on Chip	TLB Translation Lookaside Buffer
RTL Register Transfer Level	UART Universal Asynchronous Receiver/Transmitter
RTOS Real-Time Operating System	UML Unified Modeling Language
SCP Software-Configurable Processor	V2P Xilinx Virtex-II Pro
SDR Single Data Rate	VHDL Very-high-speed integrated circuits Hardware Description Language
SDRAM Synchronous Dynamic Random Access Memory	VPP Verilog PreProcessor
SI Système international d'unités	WAR Write After Read
SoC System on Chip	WAW Write After Write
SPARC Scalable Processor Architecture	WE Write Enable
SRAM Static Random Access Memory	WP Write Prepare
SSA Static Single Assignment form	

1. Introduction

Since the beginning of the microelectronic revolution with the invention of the integrated circuit in 1959 and the first microprocessor in 1971, the everlasting quest for computing power has not slowed. The demand has been satisfied and even fueled by Moore's Law, which around the same time correctly predicted a doubling of the transistor count every two years. Recently, however, the steep increase in logic cells has not resulted in equivalent gains in processing power. The excessive pipeline complexity of the single microprocessor core, which became increasingly inefficient in accelerating computation, was identified as the main reason for this stagnation [Colw04], another being the integration of larger cache hierarchies into the processor die, which only indirectly supported actual computation. As a countermeasure, not only became the single processing core more complex, but the number of cores was increased as well. Again, the performance gain did not scale proportionally to the number of cores as expected, due to the varying parallelizability of algorithms in both application programs and Operating Systems (OSs).

Hence, to leverage locality for both algorithms and data, the latest trends diversify the computing tasks into small nodes, which process data in the place where it is produced or consumed. When integrated on a single chip, such heterogeneous Systems on Chip (SoCs) often target power-constrained mobile devices, or cost-efficient low to medium scale computers. In the mobile device domain, the GreenDroid processor [GSVG10] is designed to run power-critical Android OS [Goog10] subfunctions on dedicated hardware accelerators. Targeting low to medium range computers, the Intel Sandy Bridge [Merr10a] and AMD Ontario [Walr10] SoCs pair a CPU with a graphics accelerator. The mid-range and larger scale distributed computing services are now termed *cloud computing*, which offer conventional applications, data storage, and processing. On the other hand, *ubiquitous computing* means small, special-purpose embedded systems which are deployed in a broad range of everyday commodities. In both scenarios, all services interact via networks of different structure and bandwidth, e.g., WAN (Internet), (W)LAN, or wireless sensor networks.

At the same time, the application area has broadened [Holl10]: Among the most obvious cases are consumer devices, which increasingly contain embedded systems. Besides the clearly visible personal computers, PDAs, cell phones, and electronic toys, such devices as dishwashers, washing machines, and electric cookers are also controlled by embedded systems. Television sets incurred the most notable metamorphosis from cathode ray tube-based purely analog radio frequency receivers to flat screen all-in-one computers with digital HDTV reception. Beyond domestic applications, embedded systems found their way into traffic management and systems including vehicles, trains, and aircraft, as well as into facility management (e.g., environmental, safety, and intrusion detection sensor nodes, air condition). If applied to private homes, the latter is often called *home*

1. Introduction

automation, which, beyond enhancing comfort, assists physically handicapped persons in managing daily chores. A further application area for embedded systems is electronic warehouse management, where Radio-Frequency Identification (RFID) tags are used to track the flow of goods. All building management systems can be augmented with access control, identification, and authorization schemes, where smartcards, terminals, and locks with integrated crypto processors are used for authentication.

The heterogeneous application requirements demand a large variety of computing performance and, in the case of battery-powered (mobile) devices, a limited power budget. A sensor node might run on solar power or a single battery for years, and an RFID smartcard is also scarcely powered by radio waves. On the other hand, today's smartphones are as powerful as 2000's desktop computers. The Toshiba CELL REGZA TV [Tosh10], a television set which employs a Cell Broadband Engine at 200 GFLOPS for 2D/3D image conditioning, even reaches into the realms of 90's supercomputing.

All systems share commercial constraints such as a short time to market, short market windows, and increasing Non-Recurring Engineering (NRE) costs for ASIC designs. With the shift from system on Printed Circuit Board to SoC designs, failures at the large and complex SoC ASIC designs became more probable and expensive. To avoid costly re-designs at late project phases, or even missing the market window, reconfigurable devices became eminent for prototyping. In the light of today's powerful reconfigurable devices, the question arises: Why not build real systems on reconfigurable devices instead of mere prototyping? Recent Electronic System Level (ESL) design tools support platform-based design, but require a well-defined target platform to operate on. Thus, the definition of a Reconfigurable Computing Platform becomes an enabler for ESL. As will be seen later, in conjunction with the automatic HW/SW compiler Comrade and the CoMAP tools, the Reconfigurable Computing Platform described in this work can be extended to a stand-alone ESL tool flow.

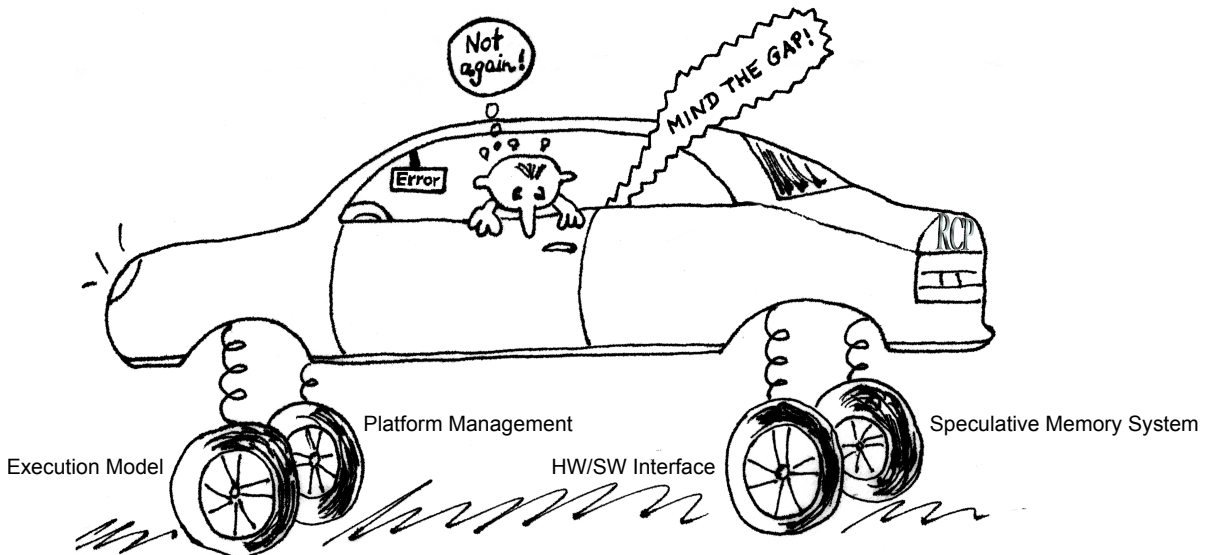


Figure 1.1.: The four Columns of a Reconfigurable Computing Platform

After introducing system architecture and tool support, the remainder of this chapter will identify four Columns that support such a Reconfigurable Computing Platform. Since cars are a good example for (in)visible networked embedded systems (e.g., entertainment, powertrain, body/ride control, safety/restraint systems), Figure 1.1 shows what happens if those Columns are not balanced properly [KCCM10, RMMT10]. To avoid such unpleasant situations, efforts are being made [FMBW09] to harmonize the rules for devices that participate in the platform. The interplay of all HW and SW components of a platform is controlled by the *execution model*. The organization process of the platform and its components is called *platform management*. The *HW/SW interface* defines a simple programming interface for both HW and SW components from a high-level language. Finally, the *speculative memory system* is designed to provide high-performance access to large data volumes while efficiently using the memory bus. The chapter is then concluded by the main contributions and the structure of this thesis.

1.1. System Architecture

Reconfigurable devices used as hardware accelerators have improved the execution speed and/or efficiency (e.g., power consumption, integration density) in many application areas (e.g., Hydra chess computer [DoLo04], molecular dynamics simulation [ChHe09], network packet processing [OhHW05, MüKo10], McEliece crypto system [SWMH09], inverse kinematics [HLSK08, LSKH09]), recent surveys can be found in [GoGr95, HaDe08].

The reconfigurable devices are engineered as either *fine-grained* architectures, such as classic FPGAs (shown in Figure 1.2) based on configurable bit-level datapaths which are built from Look-Up Tables (LUTs) and Flip-Flops (FFs) (Xilinx, Altera, Actel, Atmel), or *coarse-grained* architectures composed of configurable special purpose processing blocks (e.g., KressArray [HaKR95], Pleiades [WZGB00], MorphoSys [LSLB00], PACT XPP [BEMN03], Element CXI ECA [KBWP07]), the latter can be also organized in hierarchical processing structures (Silicon Hive [Sili03]). Despite some application-specific successes (e.g., Morpheus [VoRH09], PACT XPP [Schu09]), no broad commercial acceptance could be gained by coarse-grained devices to this day. Hence, the studies of this thesis consider the more universal fine-grained FPGAs, which have lately started to incorporate coarse-grained components such as Digital Signal Processing (DSP)/multiplier blocks, network transceivers, on-chip memory, and General Purpose Processors (GPP, an abstraction for the classic single-core CPU in a many-core environment). Notwithstanding, the findings of this work are not restricted to any specific kind of reconfigurable technology due to their system-level abstraction and general scope.

Recent developments in FPGA technology have paired a large reconfigurable logic array with powerful general purpose processor(s), either implemented as soft core within the reconfigurable fabric (Xilinx Micro/PicoBlaze [Xili10b, Xili10c], Altera Nios II [Alte10]/MIPS MP32, Actel ARM Cortex-M1/CoreMP7 [Acte10a], BEE3 RAMP [DaTC09, BWAK08], SPARC Leon [AnGW10]), or as dedicated hard core with significantly improved performance, e.g., Garp [CaHW00], Altera Excalibur (ARM922T) [Alte02], Altera Cortex-A9 [Arml10a], Stretch (Tensilica Xtensa) [Stre09], Xilinx

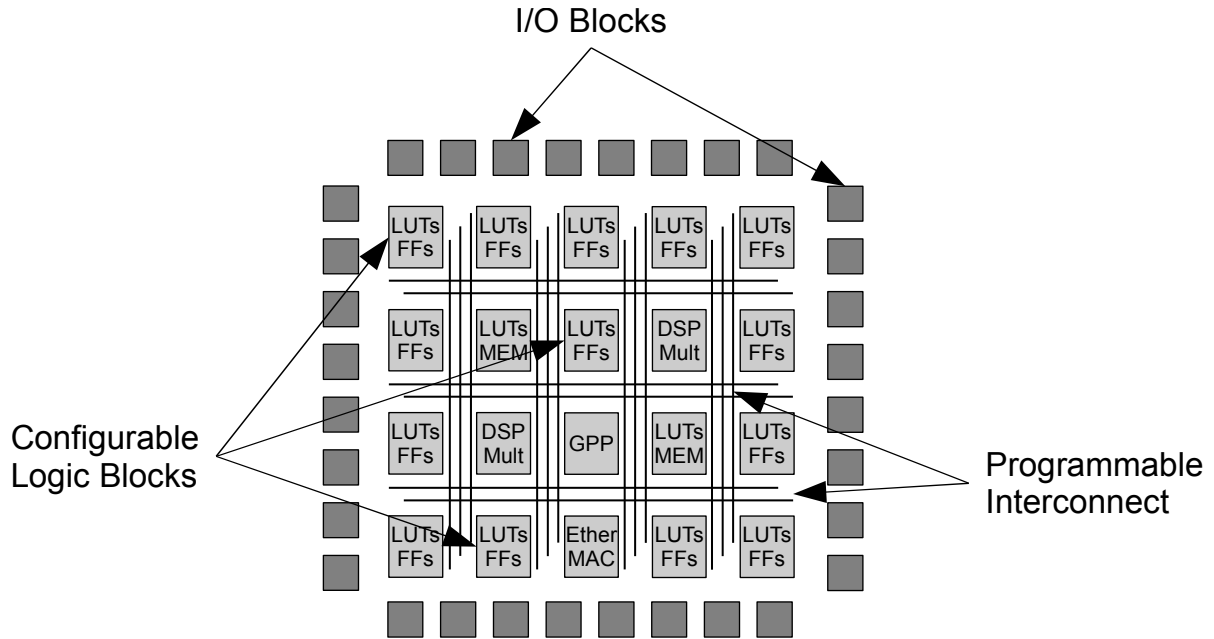


Figure 1.2.: Basic FPGA fabric architecture

(PowerPC 405/440 [Xili10f], ARM Cortex-A9Duo [Arml10a]), Actel ARM Cortex-M3 [Acte10b], and Atmel FPSLIC (AVR) [DHKM04]. The more sophisticated among these general purpose processors provide super-scalar program execution, caches running at processor core speed, and virtual memory management via an integrated or optionally attachable memory management unit. A single FPGA as reconfigurable device is at the time of writing large enough to accommodate a complete SoC including all buses, peripheral controllers, on-chip SRAM, and system logic. Even compute-intensive floating-point applications, which have traditionally been the sole reserve of dedicated DSP hardware, can now be implemented on FPGAs employing the above-mentioned integrated DSP blocks.

When implemented on an FPGA, such SoCs are often termed a *reconfigurable* SoC (an example is shown in Figure 1.3). In combination with the integrated general purpose processor(s) and optional external DDR SDRAM, the FPGA forms an entire *embedded system*. On the other hand, architectures involving combinations of a conventional processor working in tandem with an FPGA-based HW accelerator are sometimes called *adaptive computers*. Similar to adaptive computers, embedded systems combine general purpose, special purpose, and custom computing blocks. However, they do not necessarily contain a reconfigurable fabric. Nevertheless, they often benefit from (reconfigurable) hardware acceleration (e.g., the video/crypto codec of a mobile device). While classic adaptive computers essentially comprise a custom datapath, a CPU, and RAM, an increasing number of embedded systems are much more complex [Wind10] and feature larger memories, heterogeneous computing units, and peripheral interface controllers. Consequently, such elaborate hardware is complemented by an OS whose functionality exceeds that of lightweight real-time OSs, which are still used in simple

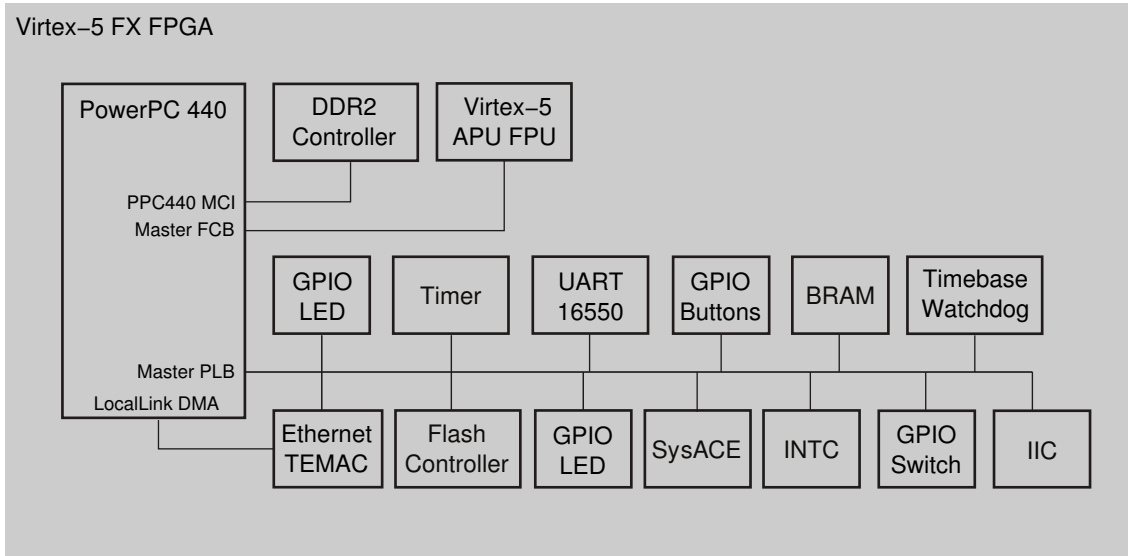


Figure 1.3.: A reconfigurable system-on-chip on a Xilinx FPGA (adapted from [Xili09a])

adaptive computers. In contrast to real-time OSs, which are often based on a streamlined C library [Xili10d, DuGV04], embedded systems increasingly run complete OSs [Tur106, VDCR08, Bala07], which comprise a kernel, device abstraction via device drivers, a virtual memory system, advanced file systems, and multitasking/multithreading. Classic adaptive computing systems as well as current reconfigurable SoCs are described in greater detail in Chapter 2.

1.2. Tool Support

Actually exploiting the potential of the reconfigurable technologies and system architectures described above, however, usually requires from the developer experience in both computer architecture and digital logic design as well as proficiency in hardware description languages.

For limited application domains (e.g., signal processing) tool flows supporting algorithm description in domain-specific languages, such as MATLAB or Simulink, have now become sufficiently mature for industrial use [Synp08, Xili09b]. In contrast, the automatic mapping of programs written in common high-level programming languages (such as C or Java) to a HW accelerator is still the subject of intense research. Even when tools accept a traditional high-level language, they often impose restrictions on specific language features (e.g., no pointers, no conditionals in loops [GGDN04, NBDH05]) or require additional user-specified annotations or idiomatic coding styles to guide the translation (e.g., the nature of memories and loop nests [Syno10a] or data streams [GSAK00]). Compilation will fail when a prohibited language construct is used or the annotations are insufficient.

As an alternative to pure HW compilation, modern automatic HW/SW compilers

1. Introduction

(e.g., Symphony C Compiler [Syno10a], Catapult [Ment10], Comrade [KaKo05], GarpCC [CaHW00], ASH [BVCG04], or Nimble [Macm01]) target hardware-accelerated embedded systems and adaptive computers by establishing a compile flow from a SW programming language such as C into both HW accelerators and SW. Here, scarce reconfigurable logic capacity can be saved by leaving non-critical code or sections unsuitable for reconfigurable devices, such as low instruction level parallelism, highly irregular control, or area-intensive floating-point, on the general purpose processor (and FPU). Thus, only those parts of a program with the highest potential for acceleration are compiled into HW. Furthermore, the software-programmable general purpose processor can act as a fallback for the compiler if it is unable to process some parts of the program for HW acceleration (e.g., due to area limitations). Instead of just aborting the translation, these parts can be compiled to software, and the user informed of the specific difficulties. The program always remains executable and can thus be migrated incrementally to fully exploit adaptive computing, with the user rewriting the problematical program sections as necessary.

1.3. Execution Model

Ideally, automatic HW/SW compilation tools partition the application between the two kinds (general purpose processor and HW accelerator) of processing elements. System-level performance is maximized when the communication overhead between the two partitions can be minimized. Generally, there are two classes of communication in such a setup. First, the data processed by the algorithms must be available to both partitions. This data is commonly located in main memory, inducing the need for high-performance high-bandwidth memory access for both general purpose processor and HW accelerator. Second, control information is exchanged between the two processing elements. Since only few data items are exchanged for this purpose, high-bandwidth transfers are not required. However, the communication latency for these exchanges now becomes a crucial factor and must be minimized. Hence, the computation model is highly dependent on how well both of these communication requirements can be met by the actual hardware.

Regardless of the adaptive computing system architecture, an automatic compiler requires the interaction between HW accelerator structures on the reconfigurable device, the software on the general purpose processor, and the entire system architecture (encompassing reconfigurable device, general purpose processor, memory, etc.) to be orchestrated using a common set of rules, which will be termed the *execution model*. In contrast to manual hardware design, where one can freely mix-and-match different computing paradigms as well as modify them for special cases, an automatic compiler only has a limited set of techniques available to realize the input algorithms. The execution model thus defines the possible solution space for compiler-generated implementations.

Hence, the first Column examined in this thesis concentrates on a high-performance execution model for adaptive computers and manually designed embedded systems using HW accelerators, as well as its efficient implementation on a current reconfigurable SoC. The execution model caters for fast data and control information exchange between the processing elements via standardized interfaces and protocols. On the HW side,

the FastLane+ high-bandwidth memory attachment for both HW accelerator and general purpose processor provides high-bandwidth shared memory access. FastLane+ is complemented by the FastPath low-latency communication channel between general purpose processor and HW accelerator. On the software side, both hardware systems are seamlessly integrated into a full-scale virtual memory, multi-tasking OS, which enables the HW accelerator to participate in transparent virtual memory data exchange with the software running on the general purpose processor, while maintaining all virtual memory advantages (e.g., demand paging, memory protection, swapping). The interplay of these three mechanisms reflects the proposed execution model.

1.4. Platform Management

Despite the rising complexity of modern embedded systems [Wind10], there is a lack of an overall system view that includes both HW and SW, which is required to define, manage, and match the platform properties and interfaces on the system level. As a consequence, a gap appears in the HW/SW co-design flow at the interface between HW and SW. The individual HW and SW sub-flows (from RTL to layout and software to binary code) themselves are quite mature with respect to tool support, but the interface between both requires significant manual effort to establish [LeFe09]. This applies even more strongly if the hardware contains IP cores, as these often feature complex functionality and interfaces. The challenges the designer has to cope with include large system-specific parameter sets that span a huge space of possible combinations, not all of them legal [IeIp10, Zell97, Thro00, AEGH10, LaRa02]. What is more, the hardware platform is mostly perceived as static, with its components and parameters defined at construction time. Since the quickly evolving embedded systems (especially when implemented as reconfigurable SoC comprising an adaptive computing subsystem) are *dynamic* platforms, the static view does not suffice. Instead, a *platform management* system is required which can describe both static and dynamic aspects of heterogeneous platform architectures (e.g., components, their design parameters, interfaces, and protocols) as well as the composition of the complex platform itself.

Thus, for the second Column supporting the Reconfigurable Computing Platform, this work focuses on the Configuration Manager for Abstract Parameterizations (CoMAP), a powerful repository with an extensible, matching tool set to operate on. The CoMAP tools are independent of hardware description or programming languages. Among other functions, they extract parameters from and (re)insert their values into several kinds of file formats and convert them into a language-independent, user legible notation. The designer may provide as part of this notation additional parameter interdependencies which are expressed as relations and automatically checked for consistency. The main focus of CoMAP is on hardware design flows based on Very-high-speed integrated circuits Hardware Description Language (VHDL) [IeVh09] and Verilog HDL [IeVe06], but its modular design enables the integration of any parameter-related language construct.

1.5. HW/SW Interface

The previously mentioned automatic HW/SW compilers (and conventional compilers in general) are adapted to or even developed and optimized for a certain static platform. Whenever the platform configuration is altered, the compiler has to be manually revised as well. As an alternative, this thesis proposes to employ the powerful platform management techniques developed here to precisely define dynamic platforms as a *compilation target* with a matching *execution model*. To mitigate the HW/SW interface gap and simplify the use of the hardware computing elements for a software programmer, such execution model must also provide for the integration of hand-crafted hardware blocks (known as Intellectual Property (IP) cores) from a software abstraction level. To this end, the complexity of these IP cores, which sometimes use sophisticated initialization procedures and interface protocols, has to be hidden from the developer by exposing normal *program function calls* as Application Programming Interface (API).

Consequently, the third Column discussed in this thesis concentrates on *HW/SW interface* design. With complex reconfigurable SoCs being a standard design style for current FPGAs, these issues are also becoming applicable to target platforms such as embedded systems and adaptive computers. On the other hand, their configurability allows a much tighter integration of IP blocks into the system at the datapath level than the comparatively coarse-grained on-chip buses used in the ASIC world. Two aspects play key roles in interface design. First, the interface functionality itself has to be partitioned between HW and SW realizations. Second, concrete interface mechanisms and protocols must be determined (e.g., physical connections, address ranges, transfer modes, device drivers, etc.). Both of these issues require the designer to explore a large design space, a time consuming and sometimes tedious task despite initial efforts at tool support [DOSG02]. The latter aspect is then highlighted in the context of embedding the use of IP cores in an ANSI C program. The case when said program is also partially compiled to HW is covered as well. The Parametric C Interface For IP Cores (PaCIFIC) establishes an automatic design flow presenting convenient, simple C interfaces (function prototypes) to a software programmer inexperienced in HW design. This approach hides the formal descriptions of IP- or platform behaviors and interface characteristics by encapsulating them together with other IP configuration data in the CoMAP repository.

1.6. Speculative Memory System

A further consequence of complex embedded systems is their *larger memory footprint* compared with simple real-time OS based systems. Both applications and OS resource management functions (e.g., process, memory, device management) require frequent memory accesses, which transfer larger data structures per access. The impact on the memory system of such operations can only partially be mitigated by demand paging and the low-latency high-bandwidth execution model mentioned above. Instead, the higher memory bandwidth required by the system can still lead to a *memory bottleneck* due to the single memory bus of the prevailing Von Neumann system architectures

[Neum45]. The latter limitation can partly be alleviated by multiple HW accelerators operating in parallel, thus creating multiple execution threads, which are coordinated by the execution model. Although automatic HW/SW compilers can parallelize common high-level language programs to a certain extent [SLLM06], manual intervention is often required for good results.

Hence, a more elaborate memory system is required to handle the increased traffic, which will be examined as the fourth Column in this work. Since multiple HW accelerator datapaths can operate in parallel, the resulting simultaneous data traffic should be served by a matching set of *multiple memory ports*. In addition to the standard caching and streaming mechanisms, such a memory system should offer *memory speculation* to further reduce the stress on the memory bus. Memory speculation, a special case of speculative program execution [KaYe05], comprises several functional levels with increasing complexity (e.g., read control, read data, write control, write data speculation, prefetching, transactional memory [HeMo93]). Theoretically, all of these features can be realized in hardware to provide for maximum throughput and software-transparent speculation management. However, even complex embedded systems are typically limited in hardware resources and power, especially in the case of mobile devices. Thus, a low implementation overhead is mandatory for the speculative memory system. To this end, this work develops and evaluates a hardware read control speculation system with prefetching, which at low resource overhead achieves roughly half of the speed-up that is gained by fully-fledged transactional memory [YBMM07, BGHS08].

1.7. Reconfigurable Computing Platform

Summarizing all four supporting Columns described in the previous sections, Figure 1.4 gives an overview of the system architecture, execution model, and design flow for the Reconfigurable Computing Platform that this work intends to define. An input C program (a) is processed by the compiler Comrade (c), which partitions it into SW (exported as C and compiled using a conventional SW compiler, e) and HW parts (as Verilog HDL). The latter can also be constructed by incorporating IP cores, which are taken from the repository (b) and automatically interfaced with the SW parts (d). The HW parts are subsequently processed by logic synthesis (f) and mapped into an FPGA. This FPGA, being an reconfigurable SoC, contains the general purpose processor (g, which executes the operating system and the SW parts of the compiled application) and the HW accelerator that corresponds to the HW parts (h). The FPGA also implements the communication channels between the HW accelerator, the general purpose processor, and external memory (k), which is reached via the speculative memory system (i).

1.8. Thesis Contributions

The research presented in this thesis highlights a broad range of aspects which must be addressed when designing and developing reconfigurable computing platforms. How-

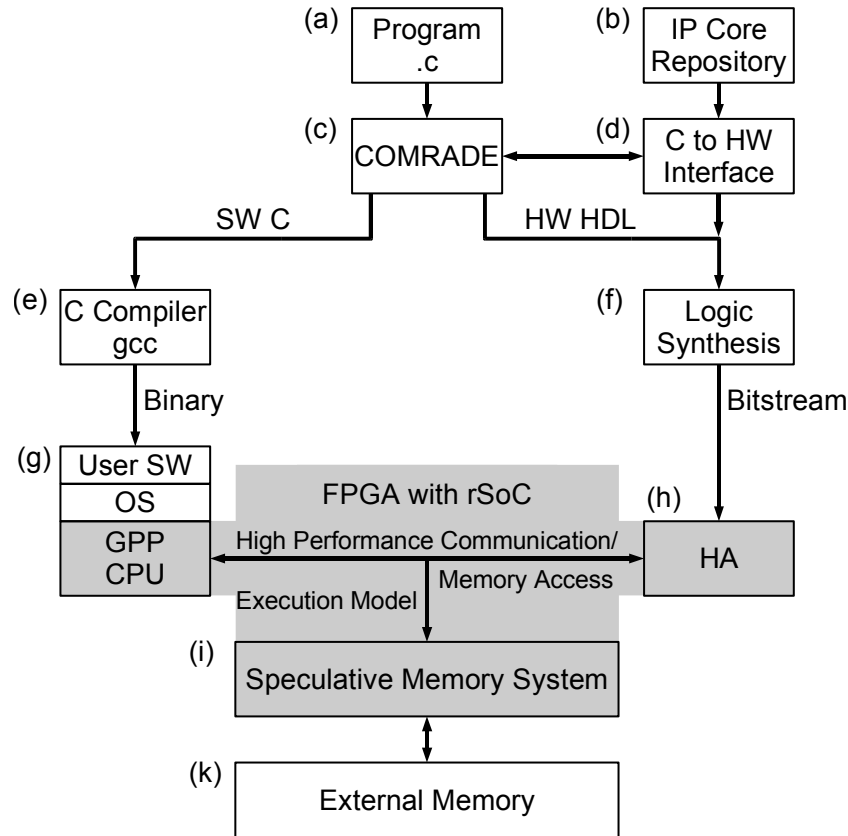


Figure 1.4.: Reconfigurable Computing Platform overview diagram

ever, most of the findings are applicable to embedded systems in general. The main contributions of this work are:

- An *execution model* which supports a fine-grained division of labor between a general purpose processor and a HW accelerator that also allows the latter to call SW functions for HW accelerator-unsuitable or infrequent operations. Actual use of this model requires certain special capabilities from a target platform. Practical solutions to all of them are presented next.
- A low-latency *communication scheme* between general purpose processor and HW accelerator that supports both efficient signaling and live variable exchange.
- High-performance *memory access* for the HW accelerator which exploits the full transfer rates of the physical memory.
- Robust, secure and efficient *system integration* of a HW accelerator in a multi-tasking protected virtual memory environment. To this end, both HW architecture as well as operating system measures are applied.
- An expressive *platform management* notation to describe static and dynamic properties of a platform, which can be stored in a dedicated repository.

- An associated set of *tools* to work on the repository and help the designer in managing large design-specific sets of parameter combinations by automatic parameter extraction from design files, and parameter set consistency checking using flexibly definable rules.
- Automatic *HW/SW interface* generation from platform properties, which also enables the integration of complex IP cores from high-level SW programming languages.
- A *speculative memory system* that offers control speculation without HW/SW program intervention by dynamically predicting the probability for the completion of memory accesses. These access statistics are exploited by two mechanisms:
- *Dynamic access priority adjustment* of concurrent memory ports that reflects their completion history, and
- *Cacheline prefetching* which prioritizes memory ports with good completion history, since their data is more likely to be accessed later.

1.9. Thesis Structure

The structure of this work follows the four Columns identified earlier in this chapter that support the Reconfigurable Computing Platform:

Chapter 1 surveys the problem area that drives this research, identifies the research problems, and presents the contributions of this thesis.

Chapter 2 introduces the terminology used in this work and reviews common HW architectures for Reconfigurable Computing Platforms.

Chapter 3 discusses the state of the art and related work in the fields of platform management, HW/SW integration, and HW architectural support for execution models, including speculative program execution.

Chapter 4 presents an execution model for Reconfigurable Computing Platforms, as well as its efficient implementation and evaluation on a commercially available reconfigurable SoC (*First Column of Reconfigurable Computing Platform*).

Chapter 5 defines a platform management system for describing platforms (e.g., the reconfigurable SoC platform used in Chapter 4) composed of components, their interfaces (which also support the execution model), as well as for describing and validating large application-specific design parameter sets (*Second Column of Reconfigurable Computing Platform*).

Chapter 6 elaborates the characteristics of HW/SW interface design and presents a C-to-HW interface represented by C functions which are automatically generated from the interface descriptions introduced in Chapter 5, and adhere to the execution model presented in Chapter 4 (*Third Column of Reconfigurable Computing Platform*).

Chapter 7 examines and discusses memory-based techniques for speculative program execution, and presents a control speculation memory system which is streamlined for the execution model shown in Chapter 4, as well as its efficient implementation on a current reconfigurable embedded system (*Fourth Column of Reconfigurable Computing Platform*).

Chapter 8 shows experimental results for all four Columns of the Reconfigurable Computing Platform, which underline the high-performance and efficient interplay of all involved parts.

Chapter 9 concludes the thesis and reflects its conceptual and experimental contributions, as well as the value of the combined four-Column approach. The discussion on the applicability of the results to related research areas is then extended to future research directions that are highlighted by the comprehensive scope of this thesis.

2. Overview of Reconfigurable Computing Platforms

While it is possible to build Reconfigurable Computing Platforms (RCPs) only from Reconfigurable Devices (RDs), it proved beneficial to combine RDs with General Purpose Processors (GPPs). Many programs spend most of their execution time in small sections of code (e.g., loop nests). The remaining statements comprise less frequent or less performance-critical operations such as I/O, memory/task management, housekeeping, or exception handling. While these operations can be implemented in HW on an RD, the marginal performance gain (if any) would not justify the expenses in reconfigurable logic area. Instead, only the computation-intense program *kernels* are implemented on the RD as a *Hardware Accelerator* (HA), all other statements are more efficiently executed on the GPP.

Adaptive Computers (ACs, a detailed discussion of ACS base architectures can be found in [Koch04]) share the same base architecture with the RCPs described in this work. However, the embedded system nature of RCPs causes further requirements, such as low power consumption, and *efficient* provision of various levels of processing performance, depending on application needs (ACs primarily aim for high performance). To this end, embedded GPPs are optimized for low power consumption and thus offer only moderate computing power. However, since all time-critical application kernels are implemented as HAs, embedded systems designed as RCPs are able to reach the computational performance of larger, power-consuming (super)computing systems at a fraction of the latter's power budgets.

With potentially many peripheral interfaces and associated heterogeneous, specialized controllers inherited from classic embedded systems, the increased complexity of RCPs over ACs has to be managed by an adequate *Operating System* (OS). The relatively simple Real Time OSs (RTOSs) are sufficient to operate ACs, which are optimized for computationally intense, yet specialized tasks, that do not require peripheral I/O or dedicated controllers. Although RTOSs are sometimes able to manage multiple threads [Oarc10], these library or microkernel-based systems often do not provide memory protection, virtual memory, or process management (e.g. LibXil [Xili10d]). Hence, the heterogeneous requirements of an RCP can only be met by a full-featured OS, which provides (in addition to process and virtual memory management) a device abstraction layer, advanced I/O services, and hierarchical file systems.

2.1. Data Exchange Modes

Beyond the OS capabilities to manage GPP and RD communication, there are architectural requirements that must be met for an efficient distributed program execution on both types of processing elements (PEs). First, it is essential that all PEs can exchange large volumes of data (e.g., images, video streams, network traffic) efficiently, since the HA performance gain is largest when many calculations process many data. While a high access bandwidth is required to source and sink the HAs and sustain a high data flow for fast computations, initial data access latencies are not harmful, since data transfers are initiated rarely and comprise larger blocks (e.g., cachelines or stream buffers). Second, whenever the execution flow is handed over to a different PE, initialization data and function parameters (known as *live variables*) must be transferred to the activated PE, and returned after execution has finished. Since only few variables have to be passed [Kasp05], a high transfer bandwidth is not required. However, the latency of these switches should be low to enable a fine execution granularity. To this end, such parameters are exchanged via *memory-mapped registers*, which GPP-PEs can access in a dedicated address range.

Such register-based access mode, which is termed *slave mode*, implies that the data transfer is initiated and controlled by a GPP-PE, while the RD-PE cannot initiate such transfers (shown in Figure 2.1 a, b, c). Although slave mode is appropriate for low-latency transfer of a small number of variables, larger data transfers would be cumbersome to handle. The RD would first have to request a transfer by signalling the GPP, e.g., via an interrupt (2.1 a). The GPP would then fetch the requested data from memory (b) and deliver it in small portions to the memory-mapped registers (c). Unfortunately, the transfer of single GPP registers often results in single cycle bus accesses instead of the more efficient burst accesses (cacheline transfers cannot be used here due to the low-latency requirement). Moreover, the RD would have to inform the GPP of every new set of addresses it needs to access, thus further slowing data delivery.

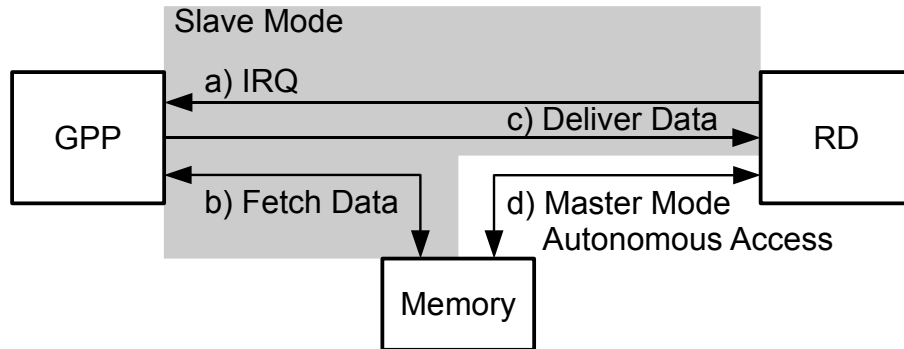


Figure 2.1.: Slave and master mode communication between GPP, RD, and memory

As an alternative, the *master mode* lifts these limitations by allowing the RD direct access to main memory (shown in Figure 2.1 d). The RD addresses the main memory autonomously, and can thus generate arbitrary access patterns at full memory speed. Hence, this *shared memory* scenario meets the above requirement for a high-bandwidth

exchange of bulk data between the PEs as well. Note that a master-mode RD also needs a slave-mode capability to receive initialization data and start the computation.

In addition to shared memory, which acts as central main memory for all PEs, each PE can have access to *local memory* that is either attached to the PE (SRAM, sometimes DRAM), or integrated with the RD-PE fabric (modern FPGAs provide up to 75 Mib of on-chip RAM). While some classes of algorithms benefit from such *memory localization* by processing many data sources and sinks in parallel, the additional overhead for managing the resulting Non-Uniform Memory Architecture (NUMA), e.g., copying data between shared and local memories to make it available to a specific PE, must be accounted for when assessing overall application performance.

2.2. RCP System Architecture

Although it is clear now that an RCP is based on RD(s) (but not necessarily FPGAs) and GPP(s), there are still several degrees of freedom to arrange these components, with external memory and peripheral interfaces, to compose an entire system architecture. The following discussion will be guided by four basic types of RCP architectures, which differ in increasing levels of PE integration with memories and system buses.

2.2.1. Standalone

The first and most loosely coupled *stand-alone* RCP architecture connects one or more RD(s) to a GPP-based system via relatively slow peripheral interfaces, forming separate subsystems. In the scenario shown in Figure 2.2, two subsystems are connected via a Controller Area Network (CAN) bus [IsoC10]. Each PE has access to local memory, the GPP memory also acts as main program memory. To initiate an RD computation, the GPP, which also runs the OS, has to copy all source data from main memory to the RD's local memory, start the computation via slave mode access to RD registers, and retrieve the result data from the local memory. The time-consuming double copying violates the shared memory requirement. Hence, moving a computation to the RD is

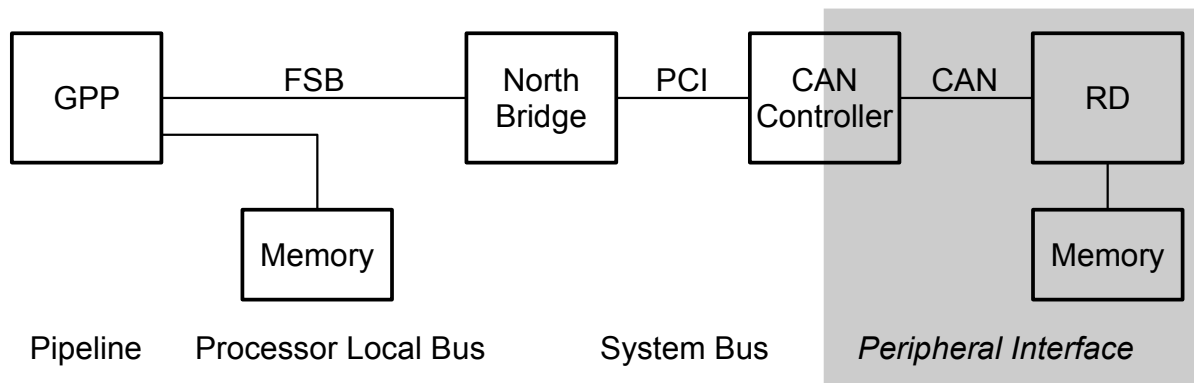


Figure 2.2.: Stand-alone RD-PE attached via CAN peripheral interface

only profitable if the HW acceleration gain is larger than the copy overhead incurred. Thus, the standalone architecture should either be used for coarse-grained switching of execution between RD and GPP, when long RD computation periods can outweigh the switching overhead, or for concurrent small tasks that run in parallel on each PE without much interaction.

Due to the standard peripheral interfaces employed to connect the PEs, such platforms are relatively cheap and simple to design and build. The design flow for these platforms requires separate setups of the RDs (HW synthesis) and GPPs (SW compilation), and has to explicitly program the slow communication links. Another application area for stand-alone RCPs is large-scale ASIC prototyping. Such systems contain up to 400 Virtex-5 LX330 FPGAs with a total of 100 GiB of DDR2 local memories [Larz09, DaTC09], and are connected to host computers, which run synthesis and verification Electronic Design Automation (EDA) software tools, via external PCI-Express links [Peri10].

2.2.2. System Bus

A tighter system integration can be achieved by connecting the PEs via a faster *system bus* such as PCI or PCI-Express. Although most implementations are designed as add-on boards for a host computer, the above-mentioned stand-alone prototyping systems can be included in the *system bus* group as well, due to their external PCI-Express links which provide comparable performance. An early representative of this group was the ACE-V Adaptive Computer [Koch00] (shown in Figure 2.3). Here, a microSPARC-IIep GPP [Sunm97], clocked at 100 MHz and accessing 64 MiB of DRAM via an integrated controller, is paired with a Xilinx Virtex 1000 FPGA, which is attached to four SRAM banks (one MiB each) of local memory. The ACE-V is designed as an add-on PCI card that requires a host for I/O and access to mass storage. The PEs are interconnected by internal PCI buses as well. To simplify the FPGA attachment, it employs a simpler bus protocol (similar to the Intel i960 Local Bus) provided via a PLX 9080 PCI bus bridge [PLXT98].

On the ACE-V, the GPP runs application SW on the RTEMS [Oarc10] OS, an

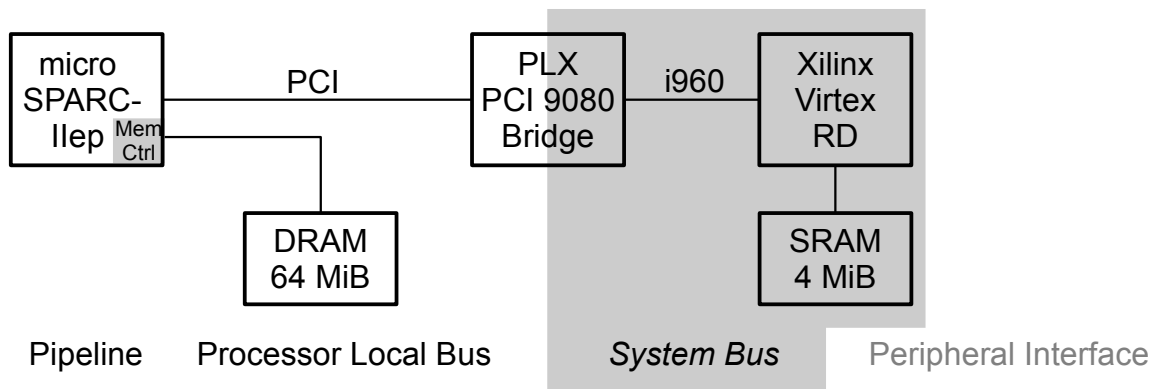


Figure 2.3.: ACE-V adaptive computer with RD-PE attached via i960 system bus

RTOS streamlined for embedded systems. Thus, the system can execute applications independently of the host PC as long as no external I/O is required. The SW part of an application is compiled using a standard C compiler and then loaded and executed on the GPP. On the other hand, FPGA HW designs are synthesized using the Xilinx ISE tools, and optionally stored in the DRAM before being programmed into the FPGA. Moreover, the ACE-V acted as the first target for the automatic HW/SW compiler Comrade [Kasp05], and was used as a prototype evaluation system for the NIMBLE compiler [Macm01]. However, the PCI buses induce a high latency (> 40 clock cycles) into live variable communication, and proved generally too slow for high-bandwidth data transfers. Although the FPGA can access the DRAM in master mode, data has to traverse two buses and a bridge. Accesses to the main memory of the host PC are even more cumbersome, as another PCI bus between GPP and host must be traversed.

A recent implementation of the system bus attachment is the Intel E600C series (Stellar-ton) [Davi10, Merr10b], which pairs an E600 SoC (Atom GPP, DDR2 SDRAM memory controller, graphics accelerator) with an Altera Arria II FPGA die in a multi-chip package. Here, GPP and FPGA are connected via two PCI-Express 1x serializer/deserializer (SERDES) links.

2.2.3. Processor Local Bus

A more modern example for system bus-attached RCPs is the Xilinx ML507 [Xili09c] embedded system development board, which is connected to the host PC via a single PCI-Express [Peri10] lane. Alternatively, it can be operated stand-alone without a host. In the latter configuration, the Virtex-5 FX FPGA, which integrates a PowerPC 440 hard core [Xili10f] (clocked at 400 MHz) with the reconfigurable fabric (100–200 MHz), accesses 256 MiB of on-board DDR2 SDRAM as well as one MiB SRAM, flash memory, I/O, and networking resources (shown in Figure 2.4). To this end, the reconfigurable System on Chip (rSoC) implemented on the RD comprises IP cores that interface with the

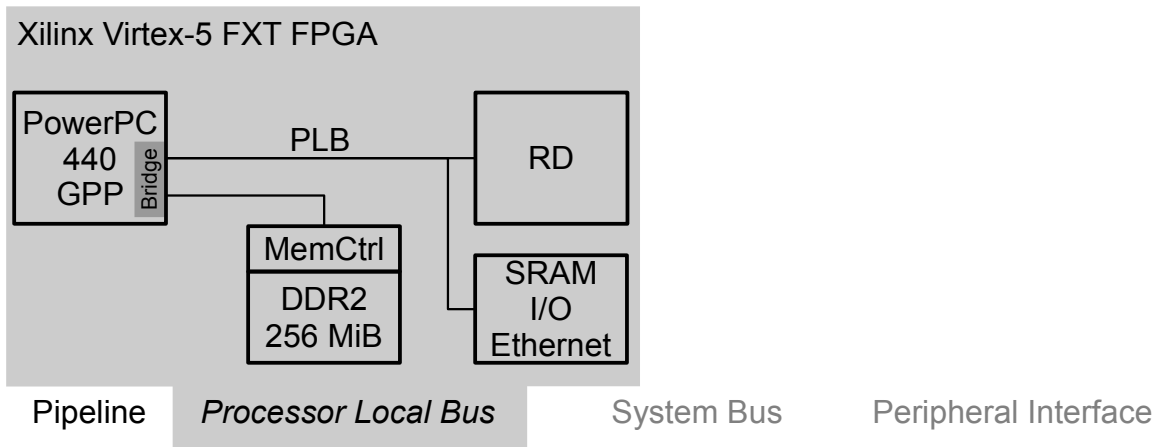


Figure 2.4.: Xilinx ML507 reconfigurable SoC with RD-PE attached via processor local bus

2. Overview of Reconfigurable Computing Platforms

peripheral logic. The GPP (PowerPC 440) and the RD are connected via the *processor local bus* (IBM CoreConnect [Inte99], 128 bit data width), which ideally provides for a tightly-coupled high-bandwidth, low-latency communication (but note the limitations of the CoreConnect implementation used here, which will be discussed in Section 4.3.2). Live variables are transferred via memory-mapped registers on the RD that are directly accessible to the GPP. Additionally, the RD can access the main memory (shared with the GPP) in master mode via the processor local bus and the memory controller. To this end, the RD has to compete with the GPP for bus access, which requires low latency arbitration and hand-over.

The GPP, although being an embedded CPU, provides HW-assisted virtual memory management, thus enabling the use of full-featured OSs (e.g., Linux). The HW abstraction provided by such OSs, however, imposes new challenges for low-latency live variable transfer and signalling, as well as for sharing memory between RD and GPP in a virtual memory environment. For instance, the memory protection offered by virtual memory as well as OS process scheduler decisions must also be obeyed by the HA, which could otherwise circumvent the protection and interfere with SW execution due to its physical access to main memory.

Due to the tight coupling of GPP and RD, which simplifies the definition of a concise yet powerful *execution model* orchestrating GPP/RD interactions, the system may be programmed using an automatic HW/SW compilation flow (e.g., Comrade, described in Section 6.5.1). However, a manual design flow involving traditional HW (register transfer level) and SW co-design can be applied as well.

The *processor local bus* class is especially suited for SoC prototyping, and with the advent of large power-efficient FPGAs, for building integrated SoCs. Hence, many different implementations are commercially available (a predecessor of the ML507, the ML310, will be described in Section 4.3). The Erlangen Slot Machine [MTAB07] consists of a PowerPC MPC875 and Virtex-2 6000 FPGA with local SRAM in a mother/babyboard configuration. The Nallatech FSB FPGA Accelerator Module [Nall10] connects to an Intel Front Side Bus (FSB) socket, replacing one CPU. It includes one or two Virtex-5 devices with two DDR2/QDR2 SDRAM banks each. A similar approach is followed by the Convey HC-1^{ex} [Conv08, Conv10], which attaches four Virtex-6 FPGAs as Application Engines (AEs) via FSB, also replacing one CPU. Host and AE memories are cache-coherently synchronized via dedicated memory controllers that additionally provide the AEs with virtual address translation. Another variant of the processor local bus attachment is the BEEcube BEE³ [DaTC09], which includes four Virtex-5 devices interconnected on a Printed Circuit Board (PCB) that can be configured with an arbitrary number of soft processor cores [BWAK08] (e.g., MicroBlaze [Xili10b]) to implement one or more reconfigurable SoCs. Among devices including ARM processor cores are the Virtex-7 [Xili10e] with a Dual Cortex-A9, and the Altera Excalibur [Alte02], which uses an ARM922T clocked at 200 MHz, embedded in an APEX 20KE FPGA. The Flash memory-based Actel Igloo [Acte10c], which can be configured with a Cortex-M1 soft core, is specialized for low power consumption.

2.2.4. Processor Pipeline

The tightest integration of the RD with the GPP is accomplished by coupling the RD directly with the *processor pipeline* (usually the Execute stage, shown in Figure 2.5). Here, the RD can directly access the GPP’s general-purpose registers, enabling very low-latency live variable transfers. To this end, no changes to the SW execution model are required, instead the RD is accessed from SW via dedicated machine instructions that extend the GPP’s standard instruction set. Thus, standard OSs for the respective GPPs continue to run without changes. However, a major disadvantage of the pipeline-driven GPP-RD communication scheme is the lack of high-bandwidth main memory access for the RD in master mode (e.g., Stretch [Stre09], PowerPC Auxiliary Processor Unit [Xili10f, Chapters 12, 15], Xilinx MicroBlaze [Xili10b], Altera Nios II [Alte10]). Hence, modern implementations enhance the RD interface with access to the pipeline’s MEM stage [JaCh99]) or dedicated memory streaming logic [Tris02, Stre09].

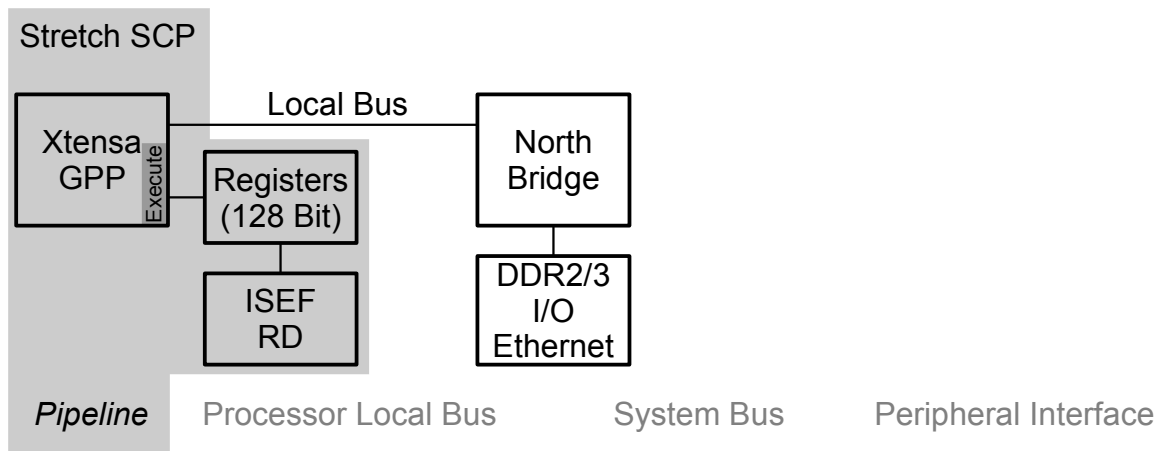


Figure 2.5.: Stretch SCP with RD-PE attached at Execute stage of processor pipeline

The Stretch Software-Configurable Processors (SCPs) use a 32 bit Xtensa pipeline as GPP, but the RD (called the Instruction Set Extension Fabric, ISEF) operates independently from and in parallel to the processor pipeline (Figure 2.5). Data is transferred between GPP, RD, and memory via dedicated 128 bit wide registers and special processor instructions which also calculate memory addresses. Recent versions of Stretch [Stre10] add an ARM9 core as service processor to run complex OSs such as Linux, and embed 64 KiB of memory directly into the RD, which are accessible in the GPP memory-map. While Stretch is programmed entirely from C/C++ using a specialized compiler that implements application-specific instructions in the RD to accelerate the SW part of the program, other *processor pipeline*-coupled systems support or require conventional HDLs for RD programming as usual.

Due to the universal yet fast *processor local bus* coupling, which allows the use of unmodified standard GPPs in contrast to the *processor pipeline* variant, platforms of the local bus type gained the largest market share. Hence, aiming for broad applicability,

2. Overview of Reconfigurable Computing Platforms

this work concentrates on the *processor local bus* type as basis for the Reconfigurable Computing Platform. The next chapter discusses prior work related to RCPs, highlighting execution models, platform management, C-to-HW interface design, and speculative memory systems.

3. Prior and Related Work

In this chapter, prior and related work will be discussed following the four Columns that support the Reconfigurable Computing Platform (identified in Chapter 1), namely the execution model, platform management, C-to-HW interface design, and the speculative memory system. Although some work relates to more than one Column (but never addresses all Columns simultaneously, as is the aim of this thesis), the respective main focus is used for classification.

3.1. First Column: Execution Model

Prior work has often considered only a limited subset of the relevant aspects. For example, [SoTB06] focuses on ease-of-use of GPP/HA communications by automatically mapping the HA registers to named files in the kernel `/proc` file system. While this removes the need for memory-mapping the hardware registers, the added file-system overhead increases latencies significantly and would impede live variable transfer.

A similar approach is described in [DLBT04], but goes further by mapping the *entire* device structure at the configuration level (CLBs, BRAMs, etc.). While this enables the exchange of large chunks of data (by allowing the GPP direct access to the HA on-chip memories), the file-system integration again leads to unacceptable latencies. None of these approaches considers HA master mode, high-bandwidth memory access, or HA memory protection and OS scheduling.

[NCVV03] proposed a message-passing interface which would allow the HA master-mode access to memory, and the underlying network-on-chip could be extended to provide an encapsulation for the OS and SW part of the application (memory protection) and enforce OS scheduler decisions (none of which is actually addressed by that work). The technique also does not deal with high-bandwidth accesses and fair memory arbitration between HA and GPP at all.

Huong and Kim [HuKi10] present an execution model that maps parts of a C program into HW at function granularity. To this end, the stack is shared between HAs and the GPP. Function calls from SW to HA and vice versa adhere to the Application Binary Interface (ABI) calling conventions of the ARM GPP employed. Average calling latencies range from 99 clock cycles for statically-linked executables to 125 cycles for dynamically-linked code, which requires dynamic address resolution. The same address resolution mechanism has to be used by the HA to access global variables and data structures, however, virtual memory is not supported. Furthermore, it is not clear whether the interrupt latency is included in the latency figures given for HA-to-SW calls.

3.1.1. Architectures Combining GPPs with HAs

An exhaustive evaluation [ScSa07] of the different possibilities to connect a GPP, HA, and memory controller via CoreConnect [Inte99] highlights the impact of the bus architecture on the communication overhead and thus the overall performance of an application.

An approach aiming at tight integration of HAs with standard Symmetric Multi Processing (SMP) systems via Intel's Front Side Bus (FSB) is presented in [Nall10]. Here, one or more GPPs are replaced by accelerator FPGA(s), which implement the FSB (64 Bit, 1066 MHz, 8.3GiB/s peak bandwidth) interface logic within their Reconfigurable Compute Unit (RCU) fabric. While such an architecture can theoretically achieve low latency and high bandwidth at the HW level, both memory bandwidth (5.8 GiB/s sustained read from system memory to FPGA, 2.8GiB/s sustained write) and latencies (700 ns for first data from memory, 105 ns for single register read) lag behind the theoretical optimums. A better approach of the memory bandwidth optimum is achieved by the FastLane+ direct memory attachment described in Section 4.5. Additionally, the programming interface of the FSB accelerator module appears to be only rudimentary: A central controller per FPGA orchestrates the I/O for one or more accelerator functions that operate as slaves on the same chip. GPPs and HAs interchange data via a statically mapped and locked-down shared portion of main memory. The HA can signal the SW application running on the GPP via IRQ, which is in turn handled by callback functions.

At the time of writing, several products [Will10] that support Intel Quick Path Interconnect (QPI) are under development. In contrast to FSB, QPI enables Non-Uniform Memory Architecture (NUMA) by supporting coherent local memories at each compute node (GPP or HA), and its simplified protocol provides for lower latencies. Since no measurements on performance or latency are publically available yet, it is unknown to which degree the requirements of the RCP execution model would be satisfied on such a platform.

The Xilinx Extensible Processing Platform [Deha10], which is based on Virtex-7 FPGAs, contains as GPP an ARM Cortex-A9 dual core [Arml10a] clocked at 800 MHz. The GPP is connected to the RCU fabric via an AMBA-4 bus [Arml10b] that provides the HA with coherent master-mode access to the GPP's caches. Since no performance measurements are officially available at the time of writing, it remains unknown to which extent the RCP requirements can be fulfilled in practice by this platform despite its promising architecture.

3.1.2. Virtually-addressed Shared Memory

For a completely different perspective on the shared memory requirement, the Philips/NXP SAA 7146A/7160E Multimedia Bridges [Phil04, NXPS08], both chips are used in many DVB systems, contain a simple Memory Management Unit (MMU) which can translate virtual addresses to physical page frame addresses if supplied with a single level page table by the associated device driver. However, this scheme is severely limited: First, the page tables are not automatically updated by the OS kernel whenever page allocations change, but by the device driver (introducing additional synchronization

overhead). Second, the single level page table, stored in a dedicated 4 KiB memory page, limits the virtual address space per DMA channel to a mere 4 MiB. For the scenario of HA and GPP acting as equal peers, such a small shared memory could be realized more efficiently using the DMA buffer/AISLE approach described in Section 4.6.2. Finally, virtual address spaces in the Philips/NXP approach are *local* to each DMA channel (8 in total), complicating unified address handling even more.

NVIDIA’s GeForce 6200 graphics accelerators introduced a technology called Turbo-Cache [NVID06], which integrated an MMU with the PCI-Express [Peri10] interface. The MMU could allocate physical page frames and use them as off-board graphics memory, in addition to faster on-board RAM. ATI/AMD uses a similar technology known as HyperMemory [ATIT05]. Both approaches suffer from slow system RAM access compared with high-bandwidth on-board graphics memory.

The shared memory requirement is also addressed by the dedicated *virtual memory window* described in [VuPI04, VuPI05]. In this implementation, HA and GPP access a virtually addressed shared memory area. The GPP is signalled by the HA when the latter attempts to access a page not present in the window, causing a virtual memory window manager in the OS to *copy* the missing page(s) between shared and main memory, thus allowing the HA full memory access. However, the technique is limited by the slow execution of the frequent and time-critical address translation (four cycles per access to the *Translation Lookaside Buffer*, TLB) as well as the high copying overhead (up to 50% of the execution time). Furthermore, the described implementation limits the window size to just 16 KiB due to the small page sizes and few pages in the window, making it unsuitable for the data-intense processing often performed by HAs.

Another approach [BrSG10] that achieves full virtual memory integration attaches the HA to a host PC via AMD HyperTransport [Hype05]. To this end, the HA is co-located on a Virtex-4 100 FPGA with an open-source HTX interface core [SGNB08] and a separate TLB for virtual to physical address translation. Communication between host GPP and the HA is controlled by an OS device driver, which is also responsible for serving HA-TLB misses, page faults, and interrupt requests. Despite the tight HW integration, the system performance is limited by high communication and memory access latencies. For example, live variable transfers are implemented via kernel system calls and thus incur a high SW processing overhead. Likewise, an HA interrupt request has to pass several layers of processing, including the HyperTransport bus, the GPP, the interrupt wait queue, and the process scheduler within the kernel. Although HA master-mode accesses are enabled by several combinations of DMA engines and caching modes, the latency for a single memory read is as high as 50 clock cycles. The frequent and time-critical address translation takes four cycles per TLB look-up, thus severely limiting the maximum memory throughput. Furthermore, the HA-TLB relies on SW to resolve TLB misses, it cannot walk the OS kernel page tables independently from the GPP. This results in excessive TLB miss penalties. While the raw HyperTransport bus speed peaks at 3.2 GiB/s, the authors evaluate their architecture with an AES core that provides a mere 80 MiB/s. By using the memory subsystem at only a fraction of its theoretical capabilities, it remains unclear to what extent the performance results reflect the actual capabilities of the system. Unsurprisingly, the authors demonstrate only low

3. Prior and Related Work

HW acceleration benefits, at reasonable dataset sizes they actually measure a slow-down compared with pure SW execution.

3.1.3. Simulated Environments

[GaCo07] presents a simulation of virtual memory integration at cacheline granularity for multiple HAs. Although shared memory access for HA and GPP is provided here as well, the relatively small (16 entry) translation lookaside buffer (TLB) for the HA relies entirely on SW management, which imposes an unacceptable overhead except for small per-application virtual address working sets. Furthermore, a real world system evaluation is missing. The performance of different cache architectures is explored on the same system [GaCo09], but still only simulated.

As a side effect of implementing increasingly complex GPPs on FPGAs, the GPP memory management units (MMUs) must also be ported [Half07]. The OpenSPARC T1 uses an MMU comprising a TLB with 64 entries, which had to be reduced to 16 or 8 to fit on an FPGA [ThHa08]. HA-MMUs require less area, since they can rely on the protection mechanisms of the GPP-MMU/OS *combination* to achieve memory protection. Hence, GPP- and HA-MMUs are not directly comparable.

Instead of the GPPs themselves also GPP simulators, traditionally being pure SW implementations, can be ported to FPGAs [CNHF08]. The core parts of the hybrid simulator are HW-accelerated, including the *simulation* of the UltraSPARC-III GPP(s) and associated MMU(s). On a TLB miss (and other functions that are unimplemented in the HA), the execution flow and context are migrated from the HA back to the host PC (attached to the FPGAs), which executes the remaining parts of the simulator.

3.2. Second Column: Platform Management

Prior work has often used the term *configuration management* to describe the parameterization process or the application of different parameter sets to a parameterized HW or SW component. In addition, *platform management* addresses the composition of HW or SW platforms from (parameterized) components, and thus involves connecting interfaces. In contrast to the broader scope of ISO's definition for configuration management [IsoQ03], which also covers quality and life cycle management, this work concentrates on the technical concepts and implementations of a platform management scheme.

3.2.1. Configuration Management

As a HW configuration management method, Thronicke [Thro00] introduces a parameter information model and a parameter flow graph which aim to capture the parameterization process as a whole. Based upon this theory, a parameter description format is presented, which comprises parameters and their associated properties. A graphical parameter editor and a scheduler for parameterization tasks are available, the latter processing the

parameter flow graph as input. However, the parameter representation in XML focuses on tool compatibility and is not easily legible by humans.

The latter restrictions apply to IP-XACT as well, which is an XML based meta data format describing IP cores, platform components, their design and verification environment, configuration, and SW interface. It has been adopted both as industry [KWKS08, ZyVS08] and IEEE standard [IeIp10]. By providing multiple views of a component with different abstraction levels and use cases (e.g., implementation, verification, system integration, documentation), ESL tools can retrieve the required information to assemble and synthesize an SoC platform and its components. To this end, IP cores have to be manually packaged with an IP-XACT XML description prior to usage, which captures attributes such as physical ports, interfaces, parameters, the register map, or physical properties. Hierarchy and property reuse are supported via `extends` elements. Since IP-XACT is only a framework for configuration management data, which partially relies on other tools that interpret source code embedded within the XML description and vendor-specific extensions, its usefulness heavily depends on tool support (described later).

Zeller [Zell97] describes a SW configuration management method which applies mathematical methods such as feature logics and version sets to software (re-)engineering.

3.2.2. Platform Libraries

In addition to a platform management system, platform libraries are required to provide HW and SW components as building blocks. If these parameterized blocks provide a standardized or well-defined interface, they can then be automatically composed by a HW/SW compiler or platform builder.

A parameterized cryptographic library to enable reuse of cryptographic modules in SoC designs is presented in [ScJA99]. It uses VHDL generic, generate, and configuration constructs to configure values for cryptographic features (e.g. block length) as well as the interface size and functionality.

A fully-pipelined VHDL floating-point operator library is presented in [WaBL06], including division, square root, accumulate and (de)normalization, which is suitable for RDs with dedicated multipliers and Block RAM (e.g., Xilinx, Altera). It is parameterized to support arbitrary floating-point formats, including IEEE single and double precision [IeFl08], by setting mantissa and exponent bit widths.

The Generic Library for Adaptive Computing Environments (GLACE) [NeKo01] provides simple arithmetic operators, which are created by parameterized module generators that deliver pre-placed gate-level netlists. A similar Module Library (Modlib) [GäTK10] accounts for the progress in logic synthesis over the past decade, which now generally achieves good placement and timing closure from Register Transfer Logic (RTL) descriptions. Consequently, Modlib generates operators from parameterized Verilog modules (without placement information). While all above-mentioned libraries [ScJA99, WaBL06, NeKo01, GäTK10] contain relatively small operators, which are required by automatic HW/SW compilers to compose datapaths, Modlib can also be extended to accommodate complex IP cores. However, the description of potentially

3. Prior and Related Work

complex interface protocols and master mode memory access for the IP core is not addressed by the authors.

Analogously to ASIC IP core providers, FPGA vendors provide IP core libraries to be used with their RDs [Xili06, Xili10g, Alte09, Acte10d]. These IP cores comprise building blocks to construct rSoCs as well as dedicated, application-specific HAs. Since manually integrating such partly complex cores can be a sophisticated task which involves setting up large parameter sets, the IP libraries are accompanied by graphical platform composition tools (e.g., Xilinx Embedded Development Kit, EDK [Xili06], Altera MegaWizard [Alte09]) that assist the designer in managing rSoC configurations. Xilinx EDK composes rSoC platforms of pre-defined, parameterized IP cores (e.g., GPP, SDRAM controller, UART, USB, Ethernet MAC) and system SW (OS kernel, device drivers, user applications). Busses are also modeled by black box IP cores. Thus, explicit bus descriptions are replaced by pure point to point connections between a (real) core's bus interface, and the black box (bus) IP core, which provides as many interfaces as there are bus devices.

3.2.3. Platform-Based Design

Apart from the tools complementing FPGA devices, an increasing number of commercial ESL tools support platform-based design. Magillem IP-XACT Packager [Magi10] is part of an ESL tool suite targeting ASIC, FPGA, and system-level designs. Based on the Eclipse IDE [Ecli10], it includes interfaces to third party high-level/RTL synthesis and verification tools. The designer is supported in the extraction of IP core descriptions from VHDL, Verilog HDL, and SystemC source code. The resulting IP-XACT description is automatically checked for compliance.

While Synopsys coreBuilder (part of the coreTools suite [Syno08]) can wrap IP cores for later reuse as well, it was originally designed to work with a proprietary IP core library. In conjunction with coreAssembler (IP subsystem creation) and coreConsultant (IP configuration), the coreTools suite provides a comprehensive IP core reuse solution that also supports IP-XACT. A similar functionality is provided by the Duolog Socrates suite [Duol10], which additionally offers register file and memory map management (see next Section 3.3).

3.2.4. Parameterized Applications

Beyond platform management, certain application areas require extensive parameterization to efficiently manage all operational alternatives within the source code. Bennis et al. [BeLT09] show a reconfigurable Particle Image Velocimetry (PIV) system to measure fluid velocities. The design is parameterized to adjust to application requirements both algorithmic properties (e.g., image size and window) and architecture (RAM bit width). They employ a Virtex-5 FPGA attached to four banks of DDR2 SDRAM and a host PC via PCI-X. All image processing is done on the FPGA using the Block RAMs as intermediate buffers, while the host provides raw image data and retrieves the results.

A parameterized library and IP-XACT extension supporting synchronous data flow applications such as digital radio receivers are presented in [AEGH10]. High-level application-specific parameters are used to calculate dependent low-level VHDL module parameters. A scheduler which is driven from the extended IP-XACT XML descriptions then automatically composes the module instances into synchronous datapaths.

A heavily parameterized memory access system for reconfigurable computers including caching and streaming functions is described in [LaKo00]. The parameterization is achieved by using Verilog parameters, built-in-, and external preprocessor (Verilog PreProcessor, VPP [ChTJ04]) statements.

An approach for testing configurable processor cores (Tensilica Xtensa) is shown in [EzJo05], which highlights the additional necessity for configurable testbenches in the HW domain.

Swahnberg et al. [SvGB05] underline the importance of parameterization throughout configuration management of (software) product families, product lines, and single components. It appears that the importance of configuration management itself increases with product complexity. Among others, preprocessor statements (`#ifdef...`) and design patterns (e.g., inheritance, polymorphism, templates) are considered parameterization techniques.

3.2.5. Modeling Languages and Configuration Management

There have been attempts to combine configuration management and the Unified Modeling Language (UML) [Omgi10a, Omgi10b] to support the composition of embedded systems or (r)SoCs from (complex) IP cores and software. The Modeling and Analysis of Real-Time and Embedded Systems UML profile (MARTE) [Omgi09] extends UML with real-time constraints and embedded systems characteristics, such as component-based architectures, timing, concurrency, power consumption, and memory organization. These new features enable UML as an Architecture Description Language (ADL) for HW/SW systems. Consequently, a combination of IP-XACT and UML MARTE is described in [AMKS08]. The authors aim to solve shortcomings in IP-XACT's timing representation by embedding IP-XACT descriptions in MARTE models, at the same time raising the abstraction level of IP-XACT (and the embedded SystemC it relies on) to timed communicating processes.

Most of the approaches described in this chapter suffer from a missing general view. Either the solutions are tailored to a specific application [BeLT09, AEGH10, ScJA99, WaBL06], or their methodology is restricted to certain tools or programming resp. HW description languages [Xili06, Xili10g, Alte09, Acte10d, NeKo01]. With commercial tools [Magi10, Syno08, Duol10] driving the evolution of IP-XACT and UML-MARTE, they implement by now a superset of the functionality supported by the combined CoMAP/PaCIFIC approach that is described in this work. However, they do not provide an integral concept for SW high-level language (e.g., C) to IP core interface that covers all aspects of low-latency live variable transfer, HW/SW signalling, and high-bandwidth master mode memory access, which is essential for most IP cores (see Section 6.2).

Moreover, CoMAP pre-dates [LaRa02] all above-mentioned efforts.

3.3. Third Column: HW/SW Interface

Although platform management already caters to static interface definitions, special care must be taken for the interface between sequential SW code and concurrent HW execution, when potentially incompatible data types are involved in data transfers between both domains. Furthermore, the dynamic interface protocol specifies the order and count of the transferred data items, depending on the capabilities of the transfer channels and mechanisms involved.

3.3.1. Architecture Description Languages

Tomiyama et. al. [TGHD99] compare several Architecture Description Languages (ADLs) and determine the characterizing properties to be behavior- and structure description. They demand an explicit behavior description of processors for better compiler generation. However, they consider synthesis-based ADLs or Hardware Description Languages (HDL) neither sufficiently easy-to-use nor flexible enough for this task. Recent ADLs such as AADL [LeFe09] provide behavior, platform and component descriptions, but assume a processor-centric system design and SW architecture. Thus, dynamic integration of custom datapaths and complex IP cores (including their sometimes sophisticated interface protocols) from SW descriptions is not addressed.

Beyond register file description (see previous Section 3.2.1), IP-XACT [IeIp10] also provides behavioral SW interface definitions for HW cores by including the complete device driver SW source code (e.g., C/C++) in the interface description. However, this straightforward solution is neither portable nor does it provide a sufficient abstraction level.

Balboa [DOSG02] is a HW/SW codesign framework for system models. It abstracts IP interfaces in a two-fold intermediate layer consisting of a Component Integration Language (CIL) and the Balboa Interface Description Language (BIDL) providing automatic data type matching and interface generation. The IP behavior is implemented as C++ models. Handel-C [Agil09] is an extension to the C language with explicit parallelism, hardware data types and inter-thread communication channels based on the model of Communicating Sequential Processes (CSP) [Hoar85].

3.3.2. HW/SW Compilers

There are a number of commercial compilers [Syno10a, Ment10, Cade08, Pose06a, Auto10] that perform high-level synthesis from ANSI C/C++ or SystemC code (usually, language subsets are supported) by applying advanced code transformation and optimization techniques such as loop unrolling, if-conversion, loop flattening, and data dependence analysis. They automatically employ behavior-level scheduling and resource binding on data-intensive and control-intensive algorithms that can include large memory structures.

However, they require manual restructuring of the code to benefit from variable bit width datatypes, and to avoid large buffers between sequential functional blocks [Berk10a, Berk10b]. The code must be reorganized prior to compilation to use pipelining instead, e.g., by loop fusion or storage type annotations. All compilers rely on one or more fixed execution models (e.g., attaching the HA to the GPP pipeline [Desi05, Auto10] or system bus [Syno10a, Auto10]), which sometimes restrict efficient GPP-HA communication (e.g., Poseidon Triton Builder uses polling to query the HA [Pose06b]).

The research compiler CHiMPS [PBDM08] translates parts of an ANSI-C program into HAs located on a Xilinx Virtex-5 FPGA, which is attached to a host GPP via Intel FSB. The compiler employs a massively parallel model of computation with balanced operator pipelines constructed from a RISC-like intermediate representation (CTL). If a code block is too complex for HA implementation, or includes HA-unsuitable statements such as system calls, a MicroBlaze [Xili10b] soft processor core is instantiated at the corresponding position in the datapath, which directly executes the CTL instructions in equivalent machine code. The authors highlight the importance of low-latency inter-processor communication and high-bandwidth shared memory access. Hence, they apply a many-cache memory strategy that places a separate, specialized cache at each memory node. The decision to renounce cache coherence mechanisms for improved scalability relies on the observation that most data arrays in high-performance computing applications are independent. CHiMPS provides a rudimentary interface for simple IP cores named *custom instruction blocks*, which must adhere to the same FIFO interface that is used for CHiMPS-generated HW operators.

PACT HDL [JBPT02] is designed to compile signal and image processing algorithms expressed in C language into RTL. Due to its Finite State Machine (FSM), load-store style Intermediate Representation (IR), control flow can be handled as well. To improve performance, alternative branches can execute in parallel (predicated execution), the incorrect branch is finally ignored. Furthermore, loops that contain no loop-carried dependencies can be pipelined, however, at most a single pipelinable memory port per loop can be used. On the other hand, if reduced power consumption is desired, resources can be shared by reverse code levelization, resulting in sequential execution of parallel expressions. IP cores are integrated as external operators. To this end, properties such as inputs/outputs, data bit width, latency, and area/power have to be provided manually in an accompanying file [JoBa02]. IP core behavior description is also based on FSMs. However, it remains unclear whether the state machine descriptions can handle pipelined IP cores.

All compilers generate HAs which use interface templates to communicate with the SW running on the GPP. However, there is only rudimentary support for custom interface synthesis [Auto10, JoBa02], which is required when integrating complex IP cores with the datapath. Moreover, none of the approaches address the automatic integration of complex IP cores from the C language program description.

3.3.3. Interface Synthesis

Denali Blueprint [Dena10] generates HW, SW, verification code, and documentation for HW register files that are specified in a register description Language (SystemRDL, a functional subset of IP-XACT [IeIp10]). The compact register description, similar to the register definitions found in data sheets, replaces manual coding of lengthy address decoder and register logic as well as register access SW. Blueprint can read IP-XACT input via a preprocessor. Duolog Socrates [Duol10] also provides register file and memory map management, generating similar types of output.

A method for automatic interface synthesis between a microcontroller and peripheral devices is presented in [ChOB92]. The approach addresses both static HW properties such as buses and pin descriptions as well as dynamic interface protocol descriptions represented as SW device driver code. The authors employ *sequences* to model microcontroller slave-mode accesses to device registers, which can be memory-mapped or attached to dedicated microcontroller ports. Sequences are a textual representation of timing diagrams and thus serialize signal transitions and bind data transfers to program variables. However, they do not support control flow such as loops or conditional branching (which is left to higher-level SW application code). Simultaneous actions (e.g., signal activations) are expressed by *parallel* statements, which may be included in sequences.

3.4. Fourth Column: Speculative Memory System

Speculative program execution has been a research focus for years, albeit in the context of CPU-based von Neumann architectures. Hence, most of the research work has been done in the context of SW running on single or multiple GPPs (a survey can be found in [KaYe05]). However, as speculation management implemented purely in SW can be very inefficient, attempts have been made recently to integrate various degrees of speculation support directly into the memory system.

3.4.1. Hardware-based Memory Speculation

Franklin et al. [FrSo96] describe the Address Resolution Buffer (ARB), which holds the addresses of all in-flight (executed, but not retired to architectural memory yet) memory accesses and supports load bypassing as well as dynamically unresolved (memory address not known yet) loads and stores, which are executed speculatively out-of-order. It uses address dependence disambiguation to detect conflicts, which are resolved in SW. The buffer is n-way address associative and employs sequence numbers to track access order. However, the accesses themselves must not be speculative (i.e., neither control nor data speculation), and access reordering can only be applied within a fixed instruction window.

A Load Store Queue (LSQ) improvement using Bloom filters is shown in [SDBM03]. The Bloom filters are implemented using a single bit hash or three-bit counters in saturated arithmetic and indicate that an entry is either *probably* present in the LSQ, or *assuredly* not present. Thus, 73–98% of all LSQ searches are eliminated, which greatly improves the scalability of the centralized LSQ approach. The authors raise the question

what instruction window sizes (and hence LSQ sizes) should be considered. Moreover, the work presents a good comparison and scalability analysis of other techniques such as store/load buffers, age-indexed LSQ, and address indexed LSQ (= ARB).

Gopal et al. [GVSS98] present coherent distributed caches, which each hold speculative versions of program data. By using a decentralized structure and snooping techniques, they prevent the bottleneck of a single shared resource. However, this Speculative Versioning Cache does not scale well in terms of cache coherence traffic and area (relying on additional exclusive LSQs per memory node).

In addition to memory speculation, Speculative Multithreaded Processors [MaGT98] also leverage speculatively parallel program execution. To this end, multiple iterations of single loop are executed in parallel, forming multiple threads. Each thread is represented by a small instruction window. Value prediction and speculative memory access techniques are used to dynamically resolve inter-thread dependencies (which here correspond to loop-carried dependencies).

A good introduction to the concepts of reorder buffer, speculative/finished load/store buffer, prefetching cache, load address/value prediction, memory dependence prediction, and load bypassing/forwarding can be found in [ShLi05, Chapter 5.3]. The related concepts of address prediction and data value prediction, which both require dependence disambiguation/prediction, are also described in [KaYe05, Chapters 8, 9]. All concepts can be integrated with the memory system and provide HW support for speculative loads and stores. Johnson [John91] found that load bypassing provides a speed-up of 11–19 percent, while load forwarding gains another one to four percent. Wall [Wall91] investigates the influence of window sizes on instruction-level parallelism and shows quantitative results for branch prediction, dependence resolution, alias analysis, and register renaming.

An introduction to GPP-based branch prediction schemes can be found in [KaYe05, Chapter 3]. However, the applicability of GPP-based branch prediction to fully spatial execution models [CaHW00, Macm01, KaKo05] is limited. Many schemes rely on data specific to the GPP execution model, such as branch target addresses or instruction streams, which are not available in fully spatial computation. However, the pure predictor schemes (a comparison can be found in [Mart07]) rely only on recognized patterns in data streams and can hence be used, e.g., on per-node memory address streams in fully spatial execution.

3.4.2. Data Prefetching

Data prefetching is already well explored and can efficiently hide memory latencies. It always requires hardware support, since data is prefetched into a buffer or storage, and is hence often combined with the cache (e.g., [NeDS04, LiRB01, SWAF06]). Several advanced pointer-chasing techniques such as greedy, correlation, and content-directed prefetching, as well as the related sequential or stride-prefetching stream buffers have been proposed to achieve high coverage, accuracy, and timely availability of data (a good survey can be found in [KaYe05, Chapter 7]). All techniques require address information that must be extracted from the data stream (pointer-chasing) or access pattern (stream

3. Prior and Related Work

buffer). However, the potential benefits resulting from localized per-node memory access information are not addressed.

Recent approaches to prefetching employ more complex methods to achieve higher coverage while maintaining good accuracy. The Adaptive CZone/Delta Correlation (AC/DC) prefetcher [NeDS04, NeSm05] divides memory into concentration zones (CZones) and analyzes address patterns of consecutive memory accesses by using a global history buffer to store address deltas of recent cache misses. CZone sizes and prefetch aggressiveness are adjusted dynamically based on program phase change detection.

Lin et al. [LiRB01] introduce Scheduled Region Prefetching, which issues prefetches to blocks surrounding the addresses of recent cache misses only when the memory channel is idle. Prefetched data is then stored in an LRU cache using replacement priorities that match the predicted prefetching accuracy. The technique Delta Correlating Prediction Table [GrJN09] stores the history of each load instruction in the form of address deltas. Patterns found in this history of deltas (using delta correlation) guide data prefetching.

Spatial Memory Streaming [SWAF06] analyzes GPP instruction flows (a technique known as *code correlation*) to predict memory access patterns. However, it is not applicable to fully spatial computation since it relies on the recurrence of sequential GPP instruction patterns. [EbMP09] explore prefetching techniques in a multi-core environment. They combine compiler-assisted prefetching with runtime feedback to improve on a baseline stream prefetcher. Sendag et al. [SeLK02] combine prefetching with speculative program execution and propose a wrong path cache to store read data from misspeculated program branches. They achieve better performance than a pure victim cache by eliminating cache pollution, leveraging the prefetching effect of the mispredicted loads.

3.4.3. Hardware Transactional Memory

Besides memory prefetching, Hardware Transactional Memory (HTM) is another important class of speculation support in the memory domain. Here, all memory accesses are treated as transactions, which define the atomicity and execution order of loads and stores.

3.4.3.1. Models

Leveraging the existing cache infrastructure, [StWF05] propose to replace the LSQ with a Store-Forwarding Cache, which is supported by a memory disambiguation table that tracks the latest in-flight accesses to each address, and a store FIFO. As a main advantage over LSQs, this approach does not require any Content-Addressable Memories (CAMs), which usually impose a large HW implementation overhead. Memory accesses are treated as transactions to track program order. The memory dependence predictor *enforces* predictions: Whenever two accesses are predicted to interfere, they must not be executed out of order. Thus, recovery is only required for collisions that have not been predicted beforehand.

Porter et al. [PoCT09] employ hardware transactional memory to support speculative multithreading. Their simulation results for two and four cores show a significant performance increase. However, they do not address the high HW requirements of HTM and its poor scalability. The Memory Conflict Buffer (MCB) [GCMG94] provides transaction-based HW memory disambiguation with SW recovery. To this end, the SW compiler inserts (optional) preload and check instructions to move potentially dependent memory loads before the corresponding stores. The MCB is a two-stage set-associative structure that tracks all in-flight loads, detects address collisions with stores, and invokes SW recovery code provided by the compiler if required.

LogTM [MBMH06], LogTM-SE [YBMM07], TokenTM [BGHS08], and FasTM [LuMG09] describe increasingly sophisticated versions of log-based HTM. This HTM variation uses a SW log to track the pre-transactional or speculative state of memory accesses. Specifically, LogTM [MBMH06] suggests to store the speculative state in architectural memory and buffer the original value in the log, resulting in fast commits (which the authors assume to be the common case), and slower aborts. However, this strategy (also known as *eager version management*) is supposed to achieve a superior overall performance, since transaction aborts can be hidden within lengthy GPP pipeline squashes, which are often the result of aborted transactions. Moreover, the paper provides a good survey of HTM approaches.

LogTM-SE [YBMM07] improves on LogTM by using signatures to detect address conflicts, thus enabling virtual transactions that are compatible with OS context switches, virtual memory, and paging. TokenTM [BGHS08] enforces eager conflict detection via the abstraction of tokens. It associates n tokens with every memory block, where n is some large constant. A transaction that reads block A must acquire *one* of A 's tokens, while a transaction that writes block A must acquire *all* of A 's n tokens. By using tokens, larger transactions that contain many loads and stores (so-called *unbounded* transactions) can be sustained. FasTM [LuMG09] is an eager HTM based on LogTM-SE, however, the log is kept in the L1 cache by pinning down cachelines that hold pre-transactional values. Thus, transaction aborts can be handled quicker.

A different HTM model, Transactional memory Coherence and Consistency (TCC) [HWCC04], addresses multi-processor machines by employing separate local versions of transactional data for each processor node. To guarantee consistency between all nodes, sequential execution order of accesses is checked at transaction granularity. If violated, the transactions involved are rolled back and re-executed. On the other hand, unordered transactions do not require sequencing checks, and may execute independently of each other. Coherency is preserved by broadcasting transaction commits via the existing snooping bus, which results in high and bursty bus loads.

Beyond the transactional memory models themselves, integration of HTM into a real system requires the capability to execute system calls, exceptions, and memory-mapped I/O, which due to side effects cannot be nullified and re-executed with ameliorated data. Blundell et al. [BILM06] address these system issues by allowing at most one *unrestricted* transaction at a time, which supports system calls, exceptions, I/O, and other system tasks. Whenever an unrestricted transaction is running, other restricted transactions (multiple of which may execute simultaneously) are stalled before completing any memory

operation.

3.4.3.2. Implementations

While the results presented by the above-mentioned HTM approaches are based on simulation, there are several HW implementations of HTM. ATLAS [NCWT07] is a chip-multiprocessor comprising eight PowerPC 405 GPP hard cores embedded within the Virtex-4 FX FPGA fabric of a BEE2 board. A ninth processor runs Linux and handles system calls and exceptions for the whole system. The GPP caches were modified and implemented in the FPGA fabric to support HTM version management and conflict detection using a TCC architecture.

Configurable Transactional Memory (CTM) [KaKu07] is an FPGA implementation of HTM written in VHDL, which tailors the HTM system to application-specific needs. Thus, the overall system can benefit from area/performance tradeoffs. CTM offers multi-port memory, which is divided into transactional ports for concurrent accesses, and conventional ports that are accessed via a shared bus, intended for private memory areas. Transaction ordering is similar to TCC. However, CTM efficiently supports only small transactions (less than 10 operations). Moreover, potential collisions are resolved only when transactions commit, resulting in bursty bus traffic.

The first attempt to integrate HTM support with a commercial product was the Sun Rock processor [CCEK09a]. Besides HTM, it provides aggressive speculation techniques such as *execute ahead* in case of high-latency operations (e.g., cache/TLB misses, divide instructions), or simultaneous speculative threading [CCEK09b], which dynamically extracts Instruction-Level Parallelism (ILP) from a single thread. To this end, up to two checkpoints are used to keep the processor’s architectural state, registers, and program positions. Execution then continues speculatively, and the processor’s state can be restored to the checkpoints if speculation fails. The same mechanism is leveraged for HTM, which uses *execute ahead* in conjunction with dedicated *checkpoint* and *commit* instructions. Additionally, the target cachelines of transactional stores are locked during commits to preserve the atomicity of transactions. SST and HTM do not require memory disambiguation or reorder buffers, which would otherwise have to be implemented as large, power-intensive associative structures (e.g., a CAM).

The Azul Vega 3 [Clic09] is a 54-core Java processor that supports lightweight HTM. Up to 864 cores (i.e., 16 processors) can be managed by a specially designed Java VM to exploit massive task-level parallelism. The Java VM uses HTM based on dynamic profiling to avoid contended synchronization locks. The HTM is implemented and visible within the L1 cache only. To this end, cachelines are individually flagged as holding speculatively read or written data. The SW interface uses `speculate`, `abort`, and `commit` instructions to signal the transition to respective HTM stages. Transactions are also aborted if speculative data is evicted from the cache (the corresponding cachelines are then marked invalid). A commit clears the speculation flags, indicating that the cachelines now contain valid data. In contrast to the memory state, the register state is not retained in HW and requires SW recovery instead. The L2 cache can manage up to 24 pending prefetches. Most unmodified Java code is accelerated by less than 10 percent.

3.4. Fourth Column: Speculative Memory System

The next chapters will elaborate on the four research areas, or four Columns that support Reconfigurable Computing Platforms, that have been surveyed in this chapter by previous and related work. Furthermore, they will present answers to the questions raised. Chapter 8 experimentally evaluates the effectiveness of these solutions.

3. *Prior and Related Work*

4. Execution Model

What characteristics does an execution model for Reconfigurable Computing Platforms encompass? How can they be met in practice on a real system?

In some cases, it is possible to implement GPP(s), hardware accelerator (HA), and peripherals on a single reconfigurable device, forming a *reconfigurable System-on-Chip* (rSoC). A wide variety of interconnection schemes have been proposed (e.g., [Arml10b, Open09]) for such architectures. In context of this work, the IBM CoreConnect [Inte99] approach will be examined more closely, which is employed in the Xilinx Virtex series of system FPGAs (Xilinx Virtex-II Pro (V2P) up to the latest Virtex-5 FX and Virtex-6 architectures). These devices are often used to implement rSoCs, embedding efficiently hardwired core components such as processors, memories or DSP blocks into a flexible reconfigurable fabric. However, as will be demonstrated, the implementation of CoreConnect which is used in these devices does not exploit the memory and bus bandwidths fully, but slows down data-intense HAs. The supposedly high-performance method of attaching HAs directly to the processor local bus (PLB) for memory access, as recommended by Xilinx development tools (Embedded Development Kit, EDK) [Xili06], can in practice lead to a cumbersome high-latency, low-bandwidth interface instead.

These observations are not limited to the domain of rSoCs, but can also apply to hardwired ASIC SoCs such as the IBM Blue Gene/L compute chip [OBBG05]. While this ASIC does also rely on PLB v4 for linking the L1 and L2 caches, limitations of the PLB architecture required both overclocking the interface beyond its specification and also prevented its use as a true bus (restricting it to point-to-point connections). For such an application, the lightweight interface which will be introduced in Section 4.5 would have been more suitable (and most likely both smaller and faster).

This chapter will first describe a novel execution model and its use by the HW/SW compiler Comrade [KaKo05]. Then, a wide spectrum of issues will be discussed that must be catered for when implementing it in real adaptive computers (ACS). To this end, the following sections will address the implementation of a high-performance memory system, tailored for the hybrid RCU/GPP processing of an ACS, as well as operating system issues such as virtual memory support for the RCU and low latency GPP-RCU communication. System-level experimental evaluations demonstrate the efficiency of these techniques and illustrate the practical feasibility of the ACS concept presented in this work.

4.1. Execution Models for Hardware Acceleration

In this section, the features of four execution models for hardware-accelerated program execution are compared with regard to their flexibility in optimally exploiting the given heterogeneous hardware computing capacity. All of them are designed to support the compilation of C for the combination of GPP and HA, the latter either in the form of an RCU [CaHW00] [Macm01] [KaKo05], or a dedicated hardware block [BVCG04].

The approaches differ in the granularity of the HW/ SW partitioning. ASH [BVCG04] strictly adheres to the procedure boundaries present in the original C source code. It can thus only move entire C functions between GPP and HA. However, it is able to call SW functions from the HA (see Figure 4.1). While procedures are indeed natural partitioning boundaries, the presence of C constructs that can be implemented on the HA only with difficulty (system calls, floating point operations) leads to *entire* procedures being ineligible for acceleration, even if the problematical operations occur only rarely (if ever) during actual program execution.

GarpCC [CaHW00], Nimble [Macm01] and Comrade [KaKo05] use a finer-grained model of interaction that allows HW/SW switches even *within* a procedure. In contrast to ASH, the underlying HW/SW partitioning (not discussed in this work) is based on actual dynamic profiling data. Thus, slow-downs due to excessive HW/SW communication can be avoided despite the finer partitioning granularity (see experimental results in [KaKo05]).

Figure 4.2 shows a code fragment containing a typical HW kernel (the loop), which is surrounded by low-ILP code that is better left on the GPP. For this example, it is assumed that library functions and floating-point operations are not efficient on the HA (the `printf()` and especially the `sqrt()` computation, which is executed just once).

GarpCC, Nimble and Comrade can move just the loop to the HA, and leave the initial and trailing low-ILP parts on the GPP. They differ in their handling of the HA-unsuitable code *within* the HW kernel. All three recognize the exceptional condition $v > 10000$

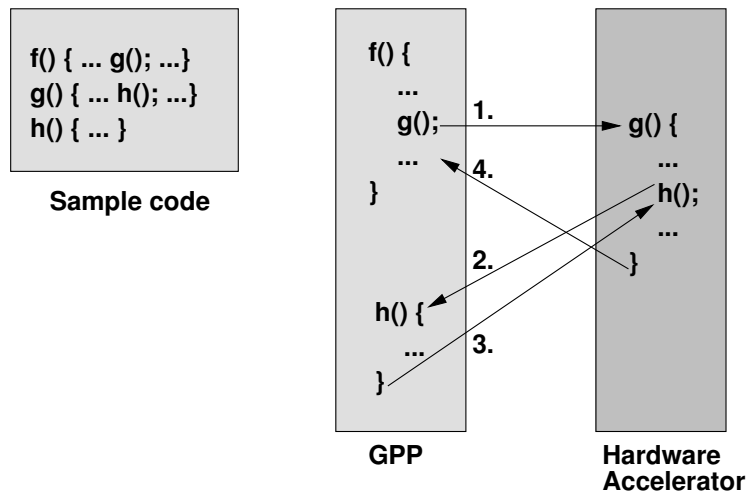


Figure 4.1.: ASH execution model

```

...
u = (int) sqrt(a + b);
v = c - d;
for (n=0; n<1000; ++n, p=p->next) {
    v += u;
    if (v > 10000) {
        printf("warning: v too large, rescaling");
        v *= 0.271844;
    }
    p->val = v;
}
w = 53 * v;
...

```

Figure 4.2.: Sample program with HA-unsuitable statements

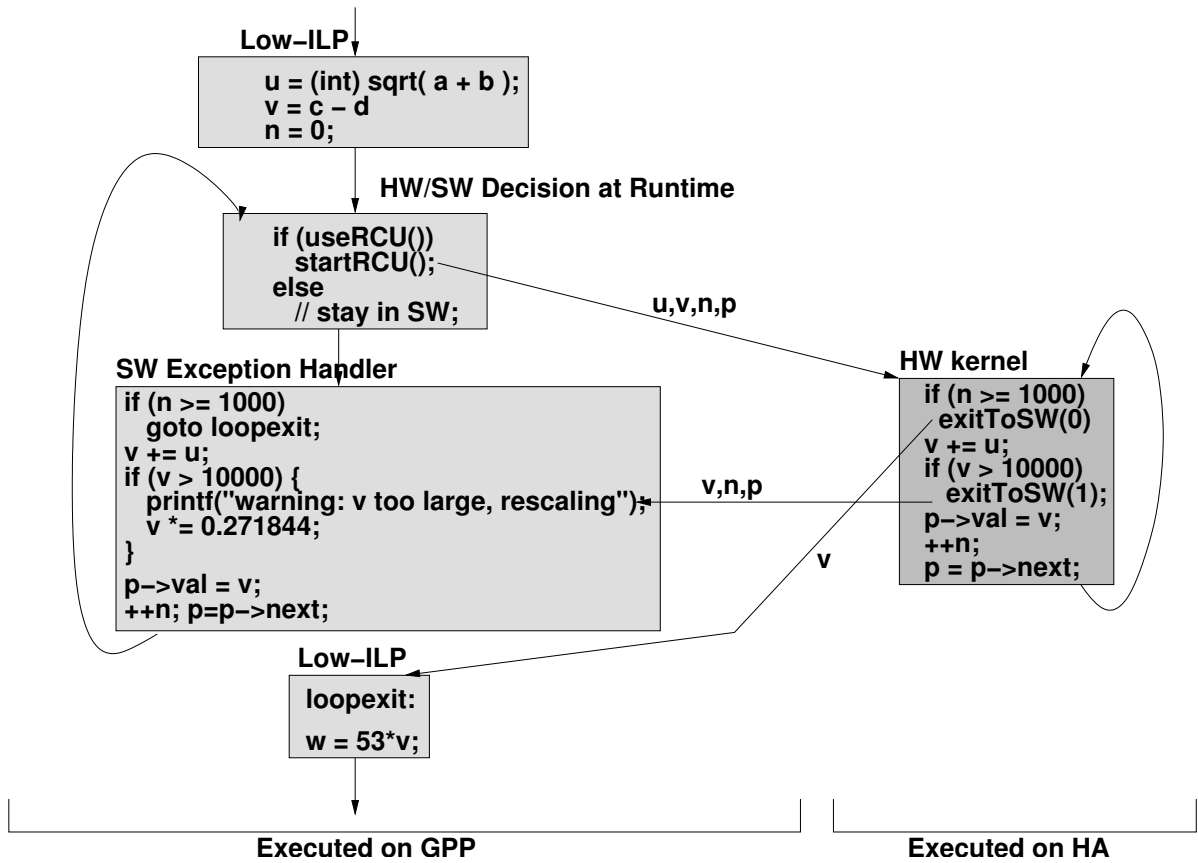


Figure 4.3.: Example in the Nimble execution model

4. Execution Model

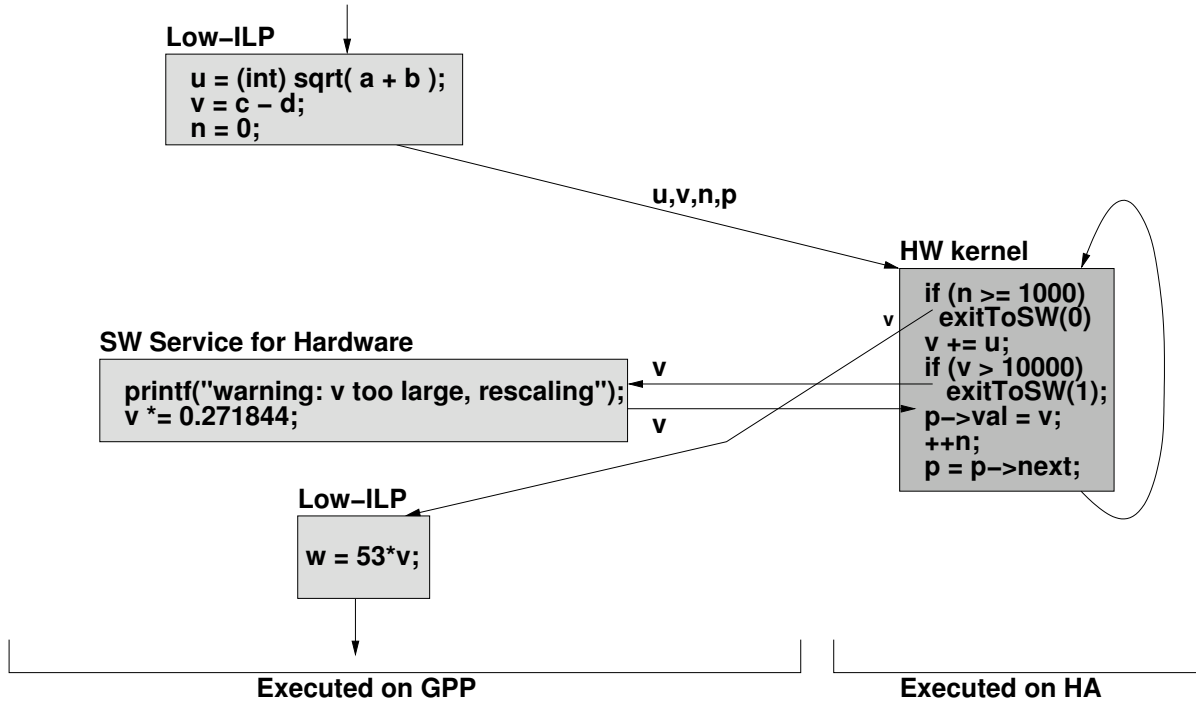


Figure 4.4.: Example in the Comrade execution model

(which presumably occurs sufficiently rarely in the dynamic profiles that moving the loop to the HA is still profitable) and execute it on the GPP. To this end, they transfer the live variables (shown as edge labels in Figures 4.3 and 4.4) from HA registers to their corresponding SW variables under control of the GPP (the HA is acting in *slave-mode* here). However, GarpCC and Nimble then execute the *entire* remainder of the current loop iteration in SW. Only when the start of the loop is reached again, is the decision made whether to run the next iteration in HW or continue executing in SW.

The new Comrade model of execution, shown in Figure 4.4, has even finer interaction granularity. Here, the HA-unsuitable library call and the floating point multiplication execute on the GPP as a *SW Service*. After completion, the SW Service *directly* returns to the HW kernel on the HA. As before, live variables need to be exchanged between GPP and HA. However, note that in the Comrade model, *fewer* variables need to be exchanged due to limiting the scope of the SW Service to just the HA-unsuitable operations (and not an entire loop iteration). In this fashion, a single HW kernel can access multiple different SW Services, each with just enough code for the requested function.

In addition to the GPP/HA interaction, the execution models need to take the pointer-heavy nature of many C programs into account. With some parts of the programs executing on the GPP, and others moved to the HA, it is crucial that pointer-based data structures (such as the lists used in the example, trees, etc.) can be freely exchanged between the cooperative HA and GPP processing. Furthermore, when operating on pointers, the HA must be able to quickly access memory on its own without intervention of the GPP (called *master-mode*).

4.2. Platform Requirements

To guide the following discussion, individual requirements on the hardware platform for actually implementing the communication mechanisms described above will be identified as **CAPITALBOLD** to link them with their implementation in later sections of this work.

All of the execution models shown in the previous section profit from low-latency communication between the GPP and the HA (**LOWLAT**, but not necessarily high-throughput, since only few data items need to be exchanged in a good partitioning). The handling of pointers as discussed above can be achieved in a number of ways: Ideally, main memory and possibly parts of the cache hierarchy are shared in a common address space (**ADDRESS**, to allow the exchange of pointers) between GPP and HA, and allow high-throughput master-mode access (**HAMEM**). However, many real ACS platforms lack this capability. In these cases, all data to be processed by the HA has to be allocated explicitly in dedicated HA-connected memory using a custom function such as `malloc_sram()`, requiring additional implementation effort (the programmer has to manually partition the memory between the different banks). Furthermore, the latter approach is only feasible for dynamically allocated memory. It fails completely for pre-initialized and stack-allocated data, both of which can occur in C programs. Each of these variables has to be manually copied into the HA-accessible memories, making sure that all contained pointers also point within the HA-memories.

Shared memory spaces are easily realized and handled when using a flat, homogeneous memory without protection between GPP and HA (common for many embedded real-time OSs): When the HA has full access to system memory, it can easily access all C-initialized and stack-allocated data.

This proves more difficult when running the ACS under an OS with virtual and protected memory such as Linux, which is becoming more and more popular even in the embedded world [Bala07]. To integrate the HA seamlessly into such an environment, a number of additional issues needs to be addressed. For security, the HA should only access the memory of the process running the SW part of the application, other memory spaces may not be compromised (**PROTSYS**). Furthermore, to maximize the effect of protected memory, the protections should also be applied *within* the HA-accelerated process (**PROTCODE**), e.g., making the SW machine code inaccessible to the HA, preserving it from potential corruption).

The performance of the rest of the multi-tasking system (including other user processes!) should not be impeded by the operation of the HA. Specifically, the HA may not deny them access to memory when they are scheduled by the OS (**OSSCHED**). This also extends to not slowing down the SW part of a hardware-accelerated application when sharing memory with the HA (**SWPERF**). Table 4.1 shows a summary of all requirements we have elaborated in this section.

The following sections will discuss in detail how these requirements can be met on currently available hardware, specifically using as HA an FPGA-based RCU, and experimentally evaluate the impact of various design choices.

4. Execution Model

Requirement	Description
LOWLAT	Low-latency GPP \leftrightarrow HA communication
ADDRESS	Shared GPP \leftrightarrow HA address space
HAMEM	High-throughput HA memory access
PROTSYS	HA access confined to own process
PROTCODE	SW code protected from HA access
OSSCHED	HA must obey OS scheduler
SWPERF	HA may not slow down SW

Table 4.1.: Summary of platform requirements

4.3. Target Platform

Since the simulation of an entire system comprising one or more GPPs, HAs, memories, and I/O peripherals is both difficult and often inaccurate, an actual HW platform is employed to evaluate the practical impact of implementing the Comrade execution model.

The Xilinx ML310 [Xili05a] is an embedded system development platform which resembles a standard PC main board (see Fig. 4.5). It features a variety of peripherals (USB, NIC, IDE, UART, AC97 audio, etc.) attached via a Southbridge ASIC to a PCI bus, which provides a realistic environment for later system-level evaluation. In contrast to a standard PC, the GPP and the usual Northbridge ASIC have been replaced by a V2P FPGA [Xili05b], which comprises two PowerPC 405 processor cores that may be clocked at up to 300MHz. They are embedded in an array of reconfigurable logic. Thus, the “heart” of the compute system (GPPs, accelerators, buses, memory interface) is now reconfigurable and amenable for architectural experimentation. With sufficient care, this

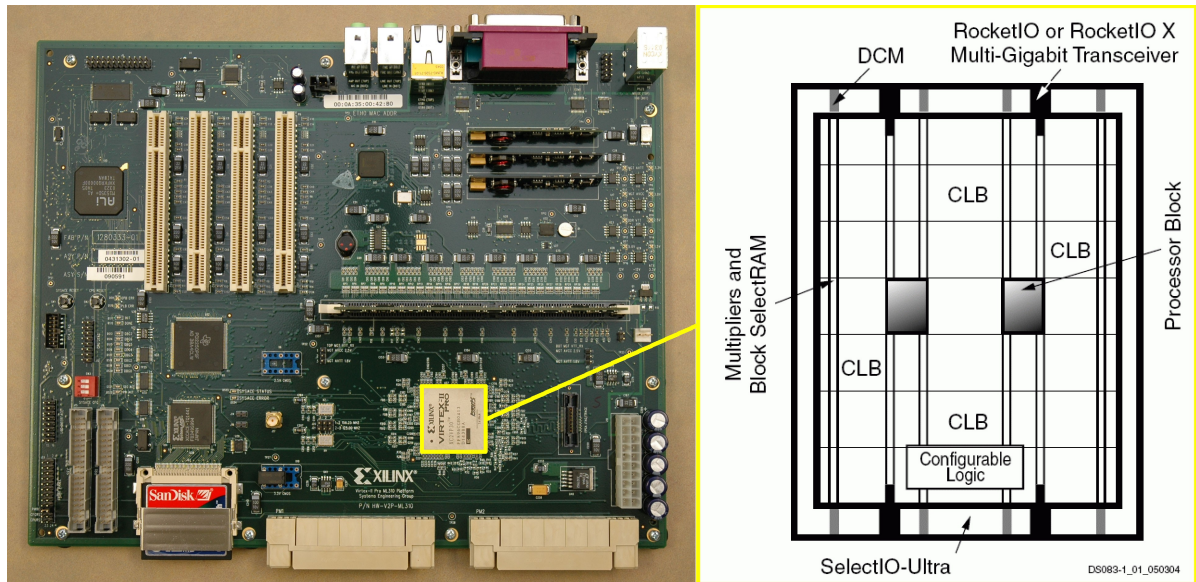


Figure 4.5.: Xilinx ML310 with Virtex-II Pro (adapted from Xilinx manuals)

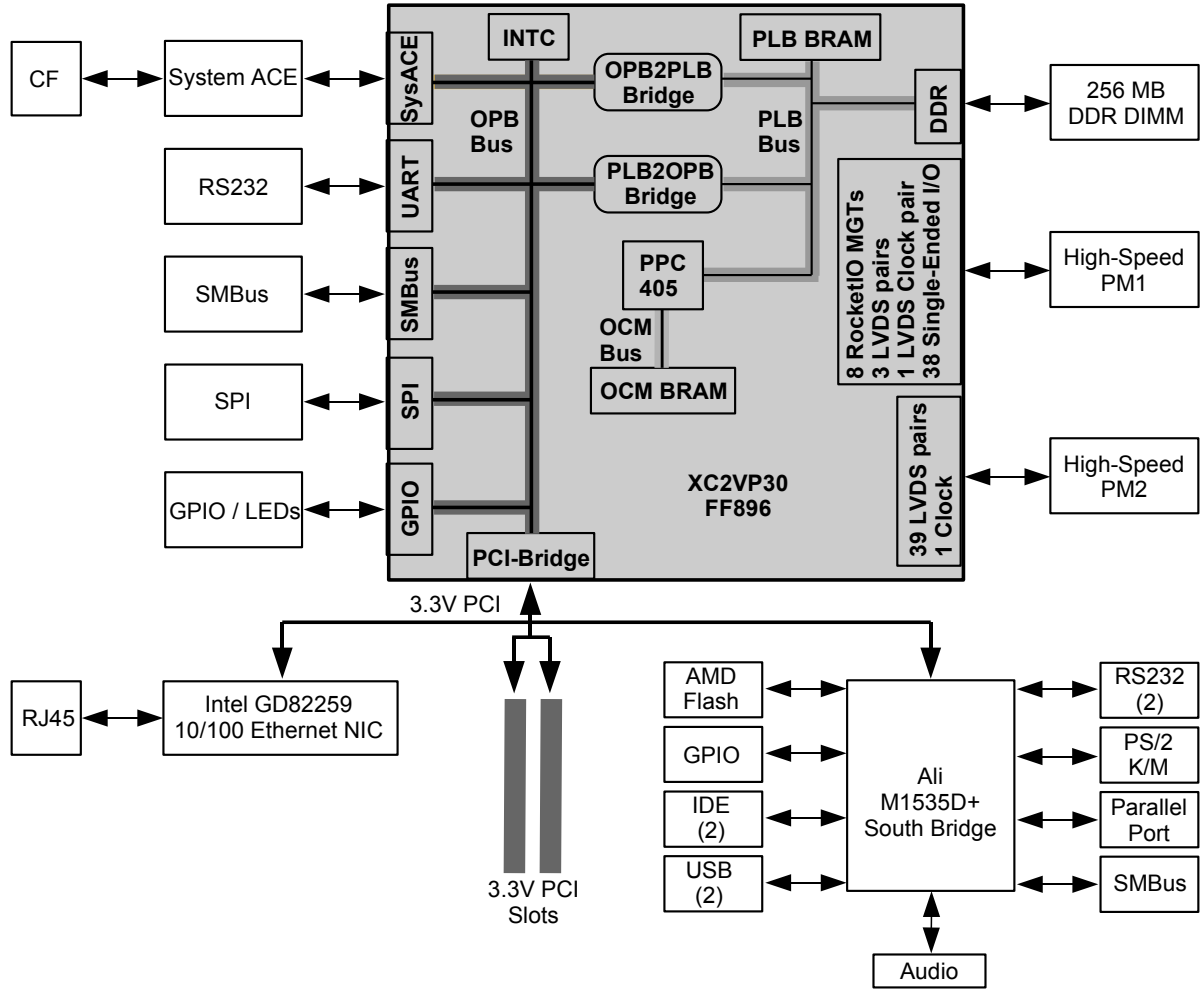


Figure 4.6.: ML310 system (from Xilinx manuals)

rSoC can implement even complex designs with a clock frequency of 100 MHz (a third of the embedded GPP cores' clock frequency).

The ML310 is shipped with a V2P PCI reference design (shown on a gray background in Fig. 4.6). This design consists of several on-chip peripherals, which are attached to a single PowerPC core by CoreConnect [Inte99] buses. These peripherals comprise memory controllers (DDR SDRAM and Block RAM), I/O (PCI-Bridge, UART, the System ACE compact flash based-boot controller, GPIO, etc.), an interrupt controller, and bridges between the different CoreConnect buses.

On the SW side of the system, standard Linux was chosen as operating system for the platform. While the use of such a full-scale multi-tasking, virtual memory system might seem overkill for the embedded area, the market share of Linux variants in that field is roughly 20% ([Turl06, VDCR08], VxWorks 9%, Windows 12%). Furthermore, Linux stresses the on-chip communication network more than a lightweight RTOS, which would impose a smaller and more deterministic load pattern on the internal buses, and is thus

better suited for load-testing the architecture under evaluation.

Note that the novel architectural concepts described in this work as well as their implementation are *not* specific to the ML310, but apply to all platforms with similar characteristics (potentially using more recent RDs, such as the Virtex-5 FX and Virtex-6 chips).

4.3.1. Vendor Flow for rSoC Composition

Regardless of the target technology, actually composing an SoC is a non-trivial endeavor. In addition to the sheer number of components (e.g., as demonstrated by the ML310), different components also have different interface requirements (e.g., an UART vs. a processor core) or may not be available using one of the supported standard interfaces at all. Efficient methods for this *platform management* are described in Chapters 5 and 6.

For the ML310 platform, standard interfaces would be either the PLB interface already mentioned above, or the simpler On-chip Peripheral Bus (OPB), which will be described below. Non-standard interfaces are common to HA blocks that often have application-specific attachments, which are then connected to standard buses by means of so-called *wrappers*. These convert between the internal and external interfaces and protocols. In some cases, the different protocols are fundamentally incompatible and can only be connected with additional latencies or even loss of features (such as degraded burst performance).

The Xilinx EDK [Xili06] SoC composition tool flow supports two standard means for integrating custom accelerators into the reference design. The simpler way is the attachment via OPB [Inte99], shown in Figure 4.7. The idea here is to isolate slower peripherals from the faster processor bus by putting them on a dedicated bus with less complex protocols. OPB attachments thus implement a relatively simple bus protocol, having 32 bits transfer width at 100 MHz clock. The most important OPB operation modes are:

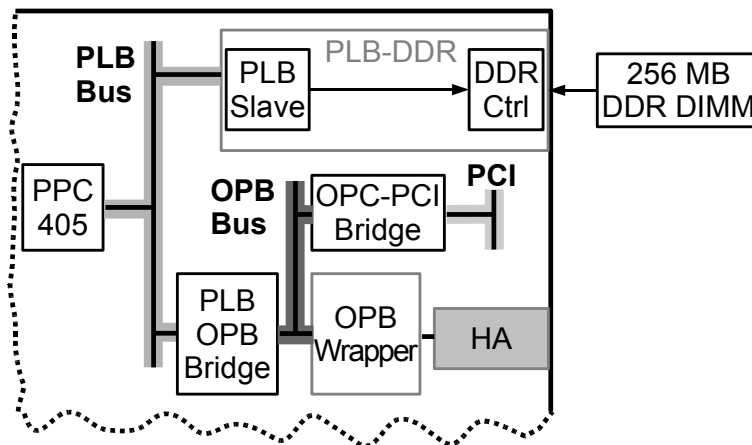


Figure 4.7.: HA integration via OPB

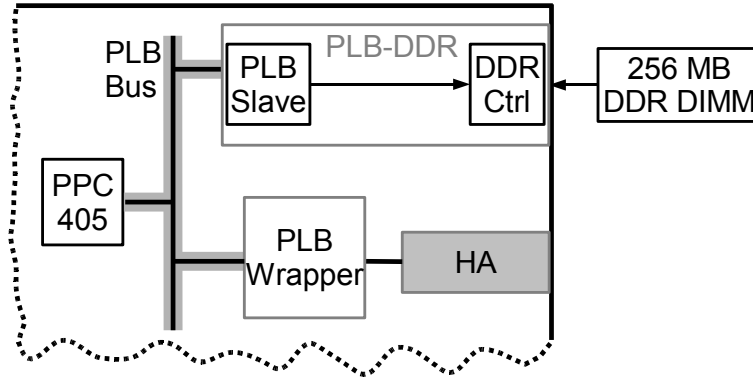


Figure 4.8.: HA integration via PLB

- Single beat transfers (one data item per transaction)
- Burst transfers of up to 16 data words (limited by processor bus)
- Master/slave (self/peer initiated) transfers

Note that even on OPB, connected blocks have master-mode capabilities in their wrappers, and can thus initiate transfers to other blocks (including the main memory) without intervention by a central processor. In the ML310, this ability is used, e.g., for the Ethernet and hard disk interfaces.

However, the simplicity comes at the price of higher access latencies compared to PLB attachment. These are due both to the OPB wrapper of the block as well as the PLB-OPB bridge which must be traversed when communicating with high-speed blocks on the PLB (such as the processor and the main memory).

The PLB attachment is the second proposed way of integrating HAs, shown in Figure 4.8. Its protocol has many features aiming for high-performance operation, coming at the cost of complexity. Hence, bus wrappers are nearly always needed when connecting to the PLB. The most important PLB operation modes are (PLB v4.6 changes used on Virtex-4 FX/-5 FX/-6 devices shown in *italics*):

- Single beat transfers (one data item per transaction)
- Burst transfers up to 16 data words
- Cacheline transfers (one cacheline in 4 *resp.* 8 data beats, cache-missed word first)
- Master/slave (self/peer initiated) transfers
- Atomic transactions (bus locking)
- Split transactions (separate masters/slaves performing simultaneous reads/writes)
- Central arbiter, but master is responsible for timely bus release (*no arbitration in point-to-point mode*)

Additionally, the PLB interface operates on 64 *resp.* 128 bits of data at 100 MHz, accompanied by access latencies from two bus wrappers between HA and DDR SDRAM controller for the main memory. Note that the controller itself also requires a wrapper. However, since the memory never initiates transactions by itself, a slave-mode wrapper suffices for this purpose.

4.3.2. Practical Limitations

As discussed in the previous section, attaching an HA to the PLB offers the highest performance possible with CoreConnect. However, there are still practical limitations: The original CoreConnect specification [Inte99] allows for unlimited PLB burst lengths, with arbitrary burst termination and deep address pipelining (if the addressed slave allows it). Unfortunately, as shown in Table 4.2, the actual implementation of the version 3.4 specification on the V2P-series of devices does not support all of the specified capabilities. The same is true for CoreConnect version 4.6, which is also shown in Table 4.2 for comparison. Again, the Virtex-5 FX/-6 implementation of this more recent revision does not leverage the full performance of PLB v4.6 (the latest PLB revision v4.7, providing DDR transactions, has not been implemented at all).

In addition to being clocked at only 100 MHz, PLB is further hindered by its relatively complex protocol and an arbitration-based access scheme, both leading to long initial latencies. The new point-to-point connection mode introduced in Virtex-5 FX/-6 only moves the problem to the interconnect crossbar of the processor block, which arbitrates the memory access requests for both GPP and peripherals. While the crossbar attaches the memory via a dedicated high performance bus (200 MHz, 128 data bits), the latter does not allow any direct connection of a peripheral due to its point-to-point nature. Furthermore, in both V2P and Virtex-5 FX/-6 Xilinx implementations, the maximum burst length is limited to just 16 words. The bus wrappers employed in the vendor tool flow for V2P impose additional latency and can also require considerable chip area (see Table 4.3). Even the performance-critical controller for the DDR SDRAM main memory is also connected to the PLB by means of a wrapper (see Figure 4.8). This combination of restrictions renders the memory subsystem insufficient for 64 bit, DDR-200 operation (1600 MiB/s theoretical peak performance, the maximum supported by the actual DDR SDRAM memory chips used on the ML310 mainboard).

	IBM PLB Spec v3.4	IBM PLB Spec v4.6	Xilinx V2P PLB
Clock	133 MHz	183 MHz	100 MHz
Address pipelining	2 cycles	unlimited	2 cycles
Latency	2 cycles	3 cycles	4 cycles
Burst length	unlimited	unlimited	16 words
Burst termination	anytime	anytime	full length

Table 4.2.: PLB specification vs. implementation

Wrapper for	Min. Size [Cells] (slave only)	Max. Size [Cells] (full master-slave)
PLB v3 64b	360	5,186
PLB v4 64b	320	3,528

Table 4.3.: Area overhead in cells (LUT+FF) for bus wrappers in vendor flow

4.4. OS Integration: Low-Latency GPP↔HA Communication

Adaptive computing systems *combine* conventional GPPs and HAs to efficiently deliver high compute performance. Comrade’s model of computation (see Section 4.1) enables the creation of HAs from code containing operations that cannot be efficiently mapped to reconfigurable logic. This might include, e.g., dynamic memory allocations or I/O such as a `printf()` function call. Normally, the presence of these operations in a computation kernel would prevent it from being realized in an HA. However, if Comrade’s hardware/software partitioning step determines that they occur sufficiently infrequently (based on dynamic profiling) the *rest* of the kernel is still realized as an HA. If these rare conditions actually arise at run-time, the HA requests execution of the hardware-infeasible operation as a SW Service on the GPP.

For high-performance, these switches should be performed with minimum latency (**LOWLAT**). This is easily achievable in an embedded system running either no operating system or only a lightweight RTOS. With the increasing complexity of embedded systems, there is a trend to run them under full-scale operating systems such as Linux [Bala07]. Unfortunately, such OSs introduce a relatively long time penalty for switching from one task to another. Interrupt handlers, which are responsible for accepting requests from hardware devices such as the HA, also suffer from additional switching delays, making hardware/software execution switches costly and violating **LOWLAT**.

As a part of this work, hardware/software mechanisms have been developed to achieve **LOWLAT** even in such a hostile environment [LaKo10]. As a baseline, consider that even when running a Linux version patched for low-latency, the interrupt response time on the ML310 platform is $62\mu\text{s}$. The interrupt initiated by the HA passes through numerous layers in the Linux kernel before it reaches the handler in the software-portion of the ACS application (shown in Figure 4.9a). The *top part* of the standard IRQ handler acknowledges the interrupt request, creates a tasklet, also known as *bottom part*, and immediately exits. The bottom part, however, is only started when invoked by the Linux scheduler, based on its priority scheme. When started, the bottom part performs the actual work of handling the request, in this case dequeuing the user process from its waitqueue, in which it has been waiting in suspended state for the interrupt to occur (cf. `acs_wait` in Figure 4.15). The now unblocked user process only continues processing when the scheduler again selects it for execution. This long overhead permits HA/SW switches only at a very coarse granularity (= long time intervals between switches), otherwise the total interrupt processing overhead becomes excessive.

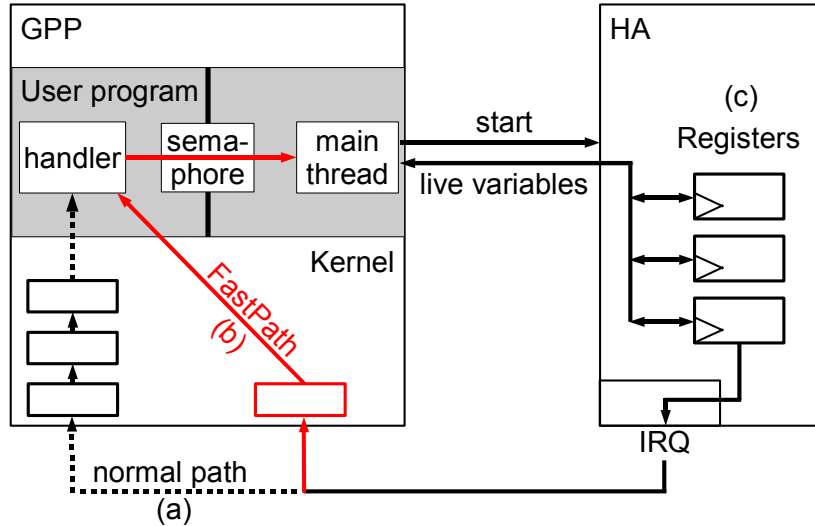


Figure 4.9.: FastPath: Low-latency SW calls and live variable transfers

To reduce the latency, the new mechanisms developed as part of this work let the HA communicate with the GPP using a *dedicated* interrupt vector of the PowerPC 405. This otherwise unused *critical interrupt* has a higher priority than the standard *external interrupt*, which signals HW-related IRQs to the Linux kernel. Nevertheless, measures have been taken to prevent IRQ priority inversion. The critical interrupt vector invokes a special handler that replies to all SW Service requests from the HA. If required, the execution flow (shown in Fig. 4.9b) can *directly* branch to a C-callback function in the user program without the two traditional scheduler interventions. Virtual addressing and access permissions are set up to allow the handler code full access to user space data (including global variables and library functions). By employing a dedicated handler, the interrupt overhead is significantly reduced, thus working towards **LOWLAT**.

The user-space interrupt handler is synchronized with the main software thread using a semaphore. This semaphore, however, is not realized using the comparatively heavyweight semaphore mechanisms in the C standard library. Instead, to avoid memory accesses and bus contention, it is implemented in a special GPP register not used by the software compiler. Thus, pressure increase on the compiler's register allocator is avoided. This GPP register (USPRG0) is reset by the main thread, which subsequently polls its status. Whenever the handler invokes the callback function, it sets USPRG0 to an arbitrary value (other than the one indicating the *reset* state), which is delivered by the HA as return code. The main thread then reads the return code from the HA invocation, resets USPRG0 again and resumes execution.

Since HA execution times are often short compared with the Linux thread scheduling overhead, the relatively short time intervals that the main thread spends polling the semaphore are negligible and a good tradeoff to avoid the thread switching overhead, which would add an unacceptable penalty to HA \leftrightarrow GPP communication latency. In fact, many modern device drivers employ a polling scheme for low-latency operation [DoTR01, ArDr99].


```

...
// get pointer to HA registers
ha = acs_get_ha_regs(NULL);

printf("ha[0] Value after RESET = %x\n", ha[0]);

// write new value
ha[0] = 0x87654321;
printf("ha[0] New value = %x\n", ha[0]);
...

```

Figure 4.10.: Fast Variable Exchange

The main thread is still controlled by the Linux scheduler, which may choose to suspend it, thus at first glance eliminating the advantage of the fast semaphore operation. While this could be avoided by increasing the priority of the main thread, such broad measures should only rarely be necessary: If operations are so latency-sensitive that they would suffer even from at most one scheduling round (which might occur during the main thread if the HA execution period exceeds the allotted time slice of the process), they can be moved to the callback handler itself and thus be executed immediately on receiving an interrupt. Remember that code in the callback has the same capabilities as in the main thread. As will be shown in Section 8.1.1, these combined measures, which are called *FastPath* in this work, significantly reduce response latency, allowing frequent hardware/software switches without increasing system load. Additionally, the interrupt response time is nearly independent of the system load. The new signalling scheme now allows the use of HAs to accelerate even shorter sections of the program. FastPath is also compatible with PaCIFIC *monolith* HA function calls (*primitive* calls do not require signalization and thread synchronization), which will be described in Chapter 6.

Beyond the quick signaling between HA and GPP, **LOWLAT** also requires a low latency data exchange between HA hardware registers and GPP software variables. This is achieved by the GPP issuing reads and writes to the memory-mapped HA registers (see Figure 4.9c). From the SW perspective, the memory mapped registers are simply accessed via a pointer to a suitable data structure. The latency for a read is 20 ns, while a write takes 40 ns per data item. Since, in general, a SW Service requires the exchange of only very few variables [Kasp05], the overhead of these data transfers is negligible compared to the signaling latency. Figure 4.10 gives an example how to pass variables between GPP and HA in slave mode. Note that the live variable exchange via memory mapped registers is also compatible with the PaCIFIC primitive and monolith HA function calls, which will be explained in Chapter 6.

4.5. High-Performance HA Memory Access

Beyond **LOWLAT**, which is only one requirement for an efficient adaptive computer, Section 4.3.2 has already shown that the combination of both PLB and Xilinx imple-

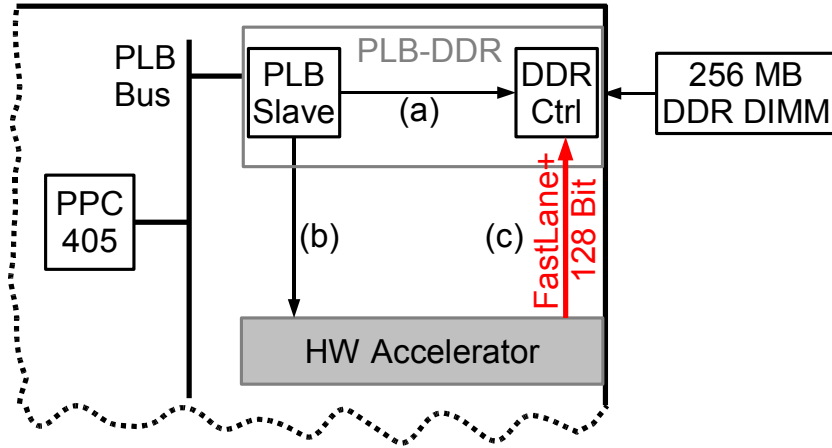


Figure 4.11.: FastLane+: Attaching HA directly to DDR controller

mentation restrictions renders the memory subsystem insufficient for 64 bit, DDR-200 operation. However, by exploiting the *reconfigurable* nature of the rSoC, an alternate architecture can be chosen. To fulfill **HAMEM**, a new approach to interface the GPP, HAs, and the main memory was designed and implemented as part of this work.

As with FastPath above, the aim is to replace generic, but potentially complex (and slow) structures with specialized, but much faster ones. The main concept behind the *FastLane* high performance memory interface [LaKo07a] is the *direct* connection of the memory-intensive HA cores to the central memory controller *without* an intervening PLB. By also using a specialized, lightweight protocol, we can avoid the arbitration as well as the protocol overhead associated with PLB. This significantly reduces the latency between the HA core and the memory controller, since no wrapper logic is interposed between the two, as opposed to two wrappers in the Xilinx reference design. We can now also make the full bandwidth of the memory controller available to the HA, eventually enabling true 64 bit double data-rate operation. Figure 4.11 shows the new memory subsystem layout.

The master-mode side of the HA is connected via FastLane directly to the interface of the DDR controller, but the PLB-side wrapper of the memory controller, which is still required for GPP memory accesses, can now *forward* GPP data transfer requests to the HA (e.g., live variables for quick HA/SW execution switches, **LOWLAT**). Thus, no additional chip area is wasted on wrappers (which have now become redundant). Furthermore, the load on the PLB signals is reduced, which improves system-wide timing. Both interfaces internally use a simple double handshake protocol, streamlined for low latency and fast burst transfers.

The original version of FastLane [LaKo07a] has since been improved by doubling its datapath width to 128 bits. The current implementation, called FastLane+, is still clocked at 100 MHz on the ML310 (200 MHz on the more recent Virtex-5 FX-based ML507 [Xili09c]), thus allowing full double data-rate operation. To this end, the existing DDR SDRAM controller of the EDK reference design [Xili06] was extended. The internal FIFO datapath transformation from the native 64 Bit DDR bus of the RAM into 64 Bit

SDR, effectively halving RAM throughput, was removed, and the full bandwidth of 128 Bit SDR (= 64 Bit DDR) was made available at the HA port.

4.5.1. Abstracting Memory Interfaces

During research in reconfigurable computing for this work, it has proven extremely useful to introduce higher-level abstractions into system architectures. Many algorithms implemented on HAs can profit from higher-level abstractions in their memory systems, such as FIFOs/prefetching for streaming data, and caches for irregular accesses. One such abstraction is the Memory Architecture for Reconfigurable Computers (MARC, shown in Figure 4.12). Instead of being focused on physical memory characteristics, as a classic memory controller would be, MARC deals with the *semantics* of memory accesses. Specifically, it provides a multi-port memory environment with separate ports for regular streaming accesses and irregular cached accesses. The HW/SW compiler Comrade uses MARC as a general memory abstraction, thus enabling the generation of HA cores which are portable between different ACS systems.

MARC, which is described in greater detail in [LaKo00], consists of three parts:

- The *core* encapsulates the functionality for caching and streaming services, the cache tag CAM (Content Addressable Memory), cache line RAM and stream FIFOs. The core also arbitrates the *back ends* and *front ends*, aiming to keep all of them working concurrently but resolving conflicts when accessing the same resource.
- The *front ends* provide standardized, simple interface ports for both streaming and caching using a simple double-handshake protocol.

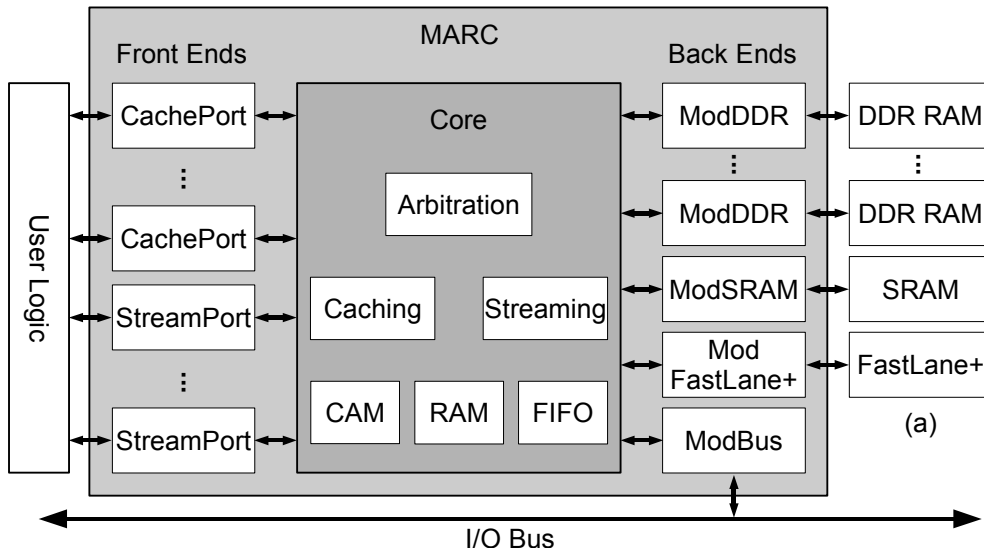


Figure 4.12.: MARC overview

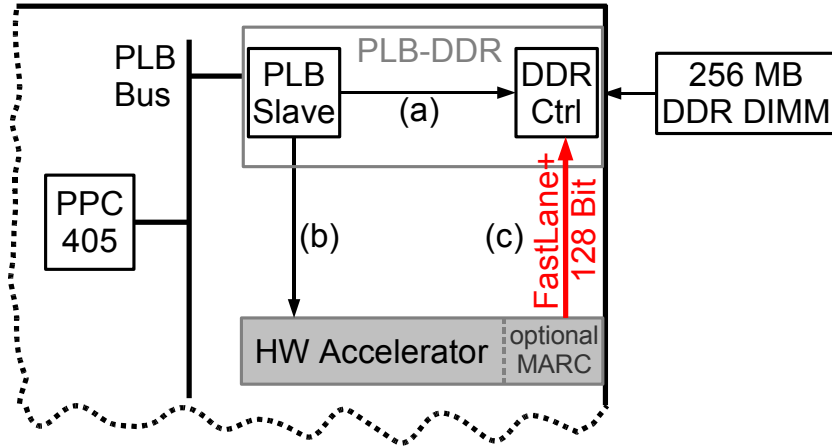


Figure 4.13.: FastLane+ with MARC interface

- The *back ends* adapt the core to several memory and bus technologies. New back ends can be easily added as required.

MARC was extended with a FastLane+ back end (Figure 4.12a) to allow both the library of manually designed HAs present at the Embedded Systems and Applications group as well as automatically compiled HAs to seamlessly benefit from the new memory system (see Figure 4.13).

4.5.2. Cooperative System Architecture

While the FastLane+ approach aims to provide optimal conditions for the compute-intensive HAs, it must also consider that the rest of the system, specifically the GPP(s), also require access to the main memory and may be intolerant to longer delays in answers to their requests. For example, interrupts, timers and the process scheduler cause memory traffic even on an idle system, and a late response leads to system instabilities. Bus master devices (capable of initiating transfers on the bus) may experience buffer over- or underruns if the transfer is not completed in time due to bus contention caused by an HA.

This implies that the GPP and other bus master devices must always have *priority* over the HA block, which can be explicitly designed to tolerate access delays. The required arbitration logic is completely hidden from the HA within the FastLane+ interface. The GPP (and other bus master devices) may interrupt master accesses of the HA at any time, while the HA cannot interrupt the GPP, and has to wait for the completion of a GPP-initiated transfer. In this fashion, scheduling decisions by the OS scheduler are enforced at the hardware architecture level, the HA can never let the GPP starve from lack of memory access (**OSSCHED**).

To this end, FastLane+ switches between two operating modes. In *passive* mode, the HA does not perform any master accesses to main memory, while GPP memory requests are passed unchanged to the DDR controller (Figure 4.11a). Optionally, the GPP may

access memory-mapped registers of the HA (cf. Section 4.4) via a dedicated address space. FastLane+ redirects these accesses transparently for the GPP to the slave interface of the HA (Figure 4.11b). On the other hand, in *active* mode, the HA is granted master access to main memory (Figure 4.11c), whereas GPP memory requests are postponed until FastLane+ switches back to passive mode. Whenever a GPP memory request is detected, FastLane+ interrupts a currently active HA master transfer as quickly as possible, providing for safe bus transaction shutdown. In contrast, HA transfers are not interrupted if the GPP accesses the memory-mapped registers of the HA, both transaction may execute in parallel. The switches between active and passive mode are controlled by a simple FSM which autonomously detects and acts upon the different operating states.

4.6. OS Integration: Virtual Memory

An adaptive computing *system* has to consider both HW and SW architectures, since, in the end, it is software applications that are to profit from hardware acceleration. Thus, the HAs must be integrated efficiently and securely with the operating system, the software environment shared by all programs running on the ACS.

Integrating an HA that is capable of independent master-mode access to main memory into an OS environment supporting virtual memory is a non-trivial endeavor. The memory management unit (MMU, see Figure 4.14) translates the virtual *user space* addresses as seen by SW applications into physical bus addresses, which are sent out from the GPP via the PLB. Address translations and the resolution of page faults are transparent for SW. Since the HAs do not have access to the MMU (integrated in the GPP) with its page address remapping tables, this implies that hard- and software communication in a virtual memory environment must use *both* virtual user space and physical addresses. Furthermore, since the HA is neither aware of virtual addresses, nor can it handle page faults, the memory pages accessed by the HA *must* be present in RAM before starting the HA.

The next sections describe three increasingly capable approaches for HA/SW interaction in a virtual memory environment.

4.6.1. Initial Approach: In-Memory DMA Buffer

The straightforward solution to this requirement would be a so-called *Direct Memory Access Buffer* (DMA Buffer). In the Linux virtual memory environment, a DMA Buffer is guaranteed to consist of contiguous physical memory pages that are locked down and always present in physical RAM, they can never be swapped out to disk. As described previously, there are now *two* addresses pointing to the Buffer, the first being the physical bus address as seen by the HA, the second being the virtual userspace address representing the same memory area for application SW. In the example given in Figure 4.14, a SW program has allocated a DMA Buffer and passes its physical address to the HA. The SW can access this Buffer via userspace address 0x01004000, which is translated to physical

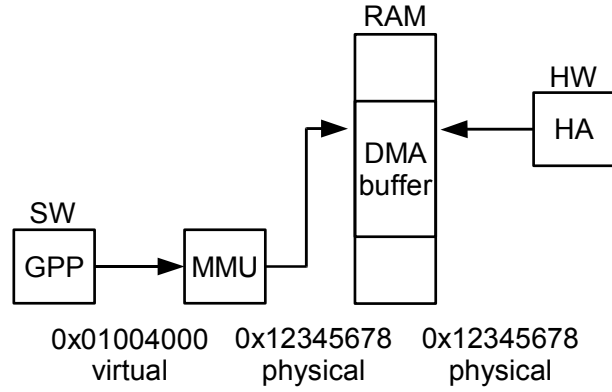


Figure 4.14.: HW and SW addressing of memory

address `0x12345678` by the MMU. The HA directly uses this physical address to access the same DMA Buffer.

Figure 4.15 sketches the interaction of HA and SW mentioned above from a programmer’s point of view. The library function `acs_malloc_master` allocates a DMA Buffer and returns both addresses. Two Buffers are allocated for input (`..._in`) and output (`..._out`), respectively. After the SW has read the input data for the HA from a file into the Buffer (addressed by its virtual userspace address `buffer_in`), the physical addresses for both input and output Buffers (`buffer_phys_...`) are transferred to the HA, which is subsequently started. The HA fetches the input data from the input Buffer via its physical address (`buffer_phys_in`) and writes the processed output data to `buffer_phys_out`). When HA execution has finished, the SW reads back the results from the output Buffer, again using its virtual address, and finally saves them to a file. The example demonstrates that the API is as easy to handle as envisioned above, but it is still different from what a pure software programmer might be familiar with, mainly due to the two addresses for one buffer.

4.6.1.1. Limitations

While already integrating the HA with the OS, and efficiently sharing main memory between SW and HA, the above approach of relying on just a dedicated DMA Buffer has disadvantages. A severe one with regard to **ADDRESS** is the need to explicitly use two *different* addresses for the same memory location. This renders it impossible to share memory pointers between HW and SW, precluding a broad range of applications from execution on HAs. Furthermore, C-initialized data, typically kept in a program’s static data segments (`.data` for non-zero initialized data, `.bss` for zero-initialized data), or dynamically allocated on stack or heap, have to be explicitly copied to and from the DMA Buffer every time before and after HA execution. These copy operations take a significant amount of time when transferring larger amounts of bulk data (violating **HAMEM**).

Another problem is that Linux DMA Buffers are generally *non-cacheable* memory

```

...
// get virtual address pointer to HA registers
ha = acs_get_ha_regs(NULL);

// request memory for input and output arrays
// retrieves both virtual and physical addresses
buffer_in =
    acs_malloc_master(NUM_WORDS*sizeof(*buffer_in),
                      (void **) &buffer_phys_in);
buffer_out =
    acs_malloc_master(NUM_WORDS*sizeof(*buffer_out),
                      (void **) &buffer_phys_out);

// fill buffer from file using virtual address
fread(buffer_in, sizeof(*buffer_in), 1, file_in);

// transfer physical addresses to HA
ha[REG_SOURCE_ADDR] = buffer_phys_in;
ha[REG_DEST_ADDR]   = buffer_phys_out;
// number of data words
ha[REG_COUNT]       = NUM_WORDS;
// start command
ha[REG_START]       = 1;

// wait for end of computation (IRQ)
acs_wait();

// write buffer to file using virtual address
fwrite(buffer_out, sizeof(*buffer_out), 1, file_out);
...

```

Figure 4.15.: API example for DMA Buffer

areas. This allows the interaction with other master-mode devices (disk or network controllers): Since the GPP cannot be sure that the DMA Buffer has not been written to “behind its back” (leading to stale GPP cache data), it avoids the inconsistency by not using a cache to access the DMA Buffers at all. With the normal use of relatively small (≈ 64 KiB) DMA Buffers employed for specific purposes (transferring disk blocks or network frames), this strategy is indeed feasible. It fails completely, however, in the ACS execution model: Here, a potentially large block of general-purpose memory has to be shared between GPP and HA. If it was marked as non-cacheable, *all* accesses by the GPP would be significantly slowed down (32b single transfers instead of 64b bursts), violating **SWPERF**.

4.6.2. Refined Solution: AISLE

The Accelerator-Integrating Shared Layout for Executables (AISLE) is a refinement of the basic DMA Buffer technique. All data areas (stack, heap and data segments) of a

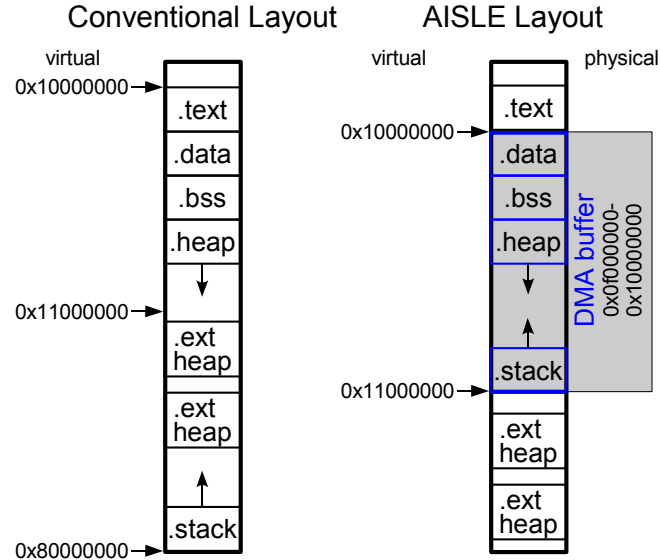


Figure 4.16.: Conventional and HA-compatible program layouts

SW executable are now kept *inside* the DMA Buffer at runtime, thus eliminating the time-consuming copy operations [LaKo07b]. Another benefit of this approach is, that pointers are now *freely interchangeable between HA and SW (ADDRESS)*: The address of every memory location as seen by the SW differs only by a *constant* offset from the address of the same location seen by the HA. This offset can be transparently removed within the HW address compute path, enabling the HA to use the *same* addresses as the SW. In contrast to conventional approaches, which rely on *explicit* communication between HA and SW to transfer data, the AISLE solution allows implicit communication: The native data structures of a program are directly shared by both HA and SW without the need to copy anything, or declare explicit HA data structures or memory areas.

To achieve this, several modifications have to be applied to the Linux kernel. The arrangement of the various areas (instructions, data, heap, stack, etc.) of a new process is established when loading the executable file from disk. Normally, when loading such a program in the common Executable and Linking Format (ELF, [TISC95]) into memory (Figure 4.16, left), the instructions in the `.text` segment as well as the `.data` segments are laid out starting from the virtual address `0x10000000`. No data is actually transferred from disk at this time, only a mapping is established from virtual addresses to the underlying disk file. Only when a virtual address is actually accessed will data be demand-paged in from disk, analogously to handling a virtual memory page fault. The same technique applies to shared library files required by the program (these are mapped-in *below* the program itself). Runtime-managed memory areas such as the heap and the stack, which have no correspondence in the program file, are mapped to anonymous memory: The heap growing upward from the end of the program, the stack growing downward from virtual address `0x80000000`. Analogous to demand paging, anonymous memory is only allocated when it is first referenced (known as *copy on write*).

The new AISLE program layout for hardware-accelerated processes in the Comrade

execution model is shown on the right side of Figure 4.16. It combines program loading with the management of a DMA Buffer (here set to 16 MiB) and deviates from the standard layout in a number of ways: First, we move the executable code *outside* of the DMA Buffer by means of a linker script. This has the effect of protecting the GPP code from rogue HA accesses (**PROTCODE**). Also, since we aim to use as small a DMA Buffer as suitable for the application (thus reducing the required address width in the HA), we also conserve buffer space in this manner. The kernel ELF loader was then altered to directly load only the data segments of AISLE programs into the DMA Buffer, which is mapped-in from 0x10000000 to 0x10FFFFFF. Specifically, we modified the function `do_mmap_pgoff` to directly load the data from the file into the Buffer, since the HA cannot use the MMU-assisted demand paging in AISLE (but see Section 4.6.3). Furthermore, `do_brk`, which extends the address space of a process for dynamically managed heap memory, was changed to hand-out DMA Buffer instead of anonymous memory (which would be inaccessible to the HA). Finally, we altered `setup_arg_pages` to initialize the user-space stack of the newly created process within the DMA Buffer, starting at the top and growing downwards. Note that all of these modifications become active *only* when loading an AISLE executable (marked by a flag in the ELF header). Conventional programs are loaded normally, fully profiting from the demand-paging mechanism.

On the hardware side, HA-internal addresses (here: 24b wide) are extended to the 32b supported by the rest of the ML310 by prepending the fixed offset of the DMA Buffer in the 32b memory space (which here requires an 8b prefix, the Buffer is always aligned with 16 MiB boundaries). Thus, even an erroneous HA cannot affect other processes (**PROTSYS**) or even the code of its own process (**PROTCODE**), accesses can occur only within the Buffer.

Since many GPPs used in embedded systems (such as the PPC405 on the V2P chip) do not have bus-snooping logic or cache coherency bus protocols (MESI, MOESI, etc.), coherency between the GPP cache and the (possibly HA-modified) DMA Buffer is maintained by SW. Before starting the HA, the control API invalidates and flushes the dirty cache lines located in the DMA Buffer out to actual memory, and invalidates all clean ones also located in the Buffer. Cache lines outside of the DMA Buffer are not affected. Once control returns to the SW process, either after the HA finishes execution or requests a SW Service, all GPP accesses to the DMA Buffer retrieve fresh data. Thus, the SW process can operate at full speed with caches enabled (**SWPERF**).

In this fashion, the AISLE-enhanced OS lets the ACS fulfill the requirements of the Comrade execution model. All of these capabilities are fully transparent to SW developers: C programs need neither explicit copy nor HA-specific memory management calls.

Figure 4.17 sketches the interaction of HA and SW from a programmer's point of view. With AISLE, the program data areas are automatically shared between HA and SW. Memory is allocated by declaring variables or using the standard C library functions in the natural way familiar to a pure SW programmer, special library function calls become redundant. The *same* address pair (`buffer_in/_out`) is then transferred to both the HA and the SW.

However, in some cases, a greater degree of control can be beneficial. As an example,

4. Execution Model

```
...
// get virtual address pointer to HA registers
ha = acs_get_ha_regs(NULL);

// memory for input and output arrays
// in standard program data area
int buffer_in [NUM_WORDS];
int buffer_out[NUM_WORDS];

// fill buffer from file using virtual address
fread(buffer_in, sizeof(*buffer_in), 1, file_in);

// transfer virtual addresses to HA
// same virtual addresses shared by HA and GPP
ha[REG_SOURCE_ADDR] = buffer_in;
ha[REG_DEST_ADDR]   = buffer_out;
// number of data words
ha[REG_COUNT]       = NUM_WORDS;
// start command
ha[REG_START]       = 1;

// wait for end of computation (IRQ)
acs_wait();

// write buffer to file using virtual address
fwrite(buffer_out, sizeof(*buffer_out), 1, file_out);
...
```

Figure 4.17.: API example for AISLE

an HA-accelerated program might profit from the allocation of large I/O data buffers that do not need to be HA-accessible themselves. While such large buffers could, of course, be realized in AISLE by simply configuring a DMA Buffer of sufficient size, this would be wasteful from a number of perspectives. First, the HA logic would require more address bits, even though it would never access the large I/O buffers to full extent. Second, since the DMA Buffer requires actual physical memory, that memory would be removed from the demand-paging virtual memory mechanisms, possibly impeding system performance as a whole. To support even these use-cases efficiently, AISLE provides *optional* API calls that allow the SW process to request *HA-inaccessible* heap memory outside of the DMA Buffer.

To this end, we use the `mallopt` function of the GNU standard C library (`glibc`), which allows to set thresholds and absolute maximum sizes for memory blocks allocated via `malloc` or `calloc`. If one of these thresholds is exceeded, `glibc` will enlarge the heap by calling the conventional `mmap` function, effectively adding a new address mapping to the heap instead of enlarging the existing one. This `mmap` request *outside* of the HA-accessible virtual address window between `0x10000000–0x1fffffff` will be served from anonymous memory as usual, leading to an *extended heap* (shown as `.ext heap` in Figure 4.16) that

stores SW-private data.

4.6.2.1. Limitations

While AISLE fulfills all requirements, it does have limitations. The size of the DMA buffer is normally set at the start of a program. At runtime, it can be resized only with significant overhead. Thus, it is always allocated to the largest data area required by the program, which can be wasteful. Additionally, DMA buffers are always present in physical memory and remove their areas from the general demand-paging performed by the OS. This not only reduces the amount of memory available to the entire system, but also forces the loading of all data areas in the program's executable file, even if they are not actually required at run-time.

4.6.3. Full Virtual Memory Support in the HA: PHASE/V

As an alternative solution, the new Processor-Hardware Accelerator Shared Environment with Virtual Addressing (PHASE/V) [LaKo08] will now be described. Here, the HA is integrated with the MMU-based virtual memory system. Instead of mapping just a *window* of the virtual address space in the form of a DMA buffer, the *complete* virtual address space is mapped between HA and GPP. All virtual memory features, such as demand paging, swap space, copy-on-write, and file-backed `mmap` mappings are thus supported both in hardware and software. Since demand paging is now available for the HA, memory pages are physically allocated (and loaded from the executable file) only when they are actually needed, not in advance.

Like many embedded processors, the PowerPC 405 does not allow external access to its MMU (Figure 4.18). Thus, a *separate* MMU had to be implemented in the HA to allow it virtual memory operations independently of the GPP MMU. This HA-MMU manages a translation lookaside buffer (TLB) of 64 direct-mapped entries which is implemented in a carefully tuned manner for the Virtex-II Pro FPGA: Both the RAM holding the physical address translations and the tag-RAM are composed of Look-Up Tables (LUTs) configured as RAM (see Figure 4.19). These Virtex library primitives provide single-cycle read operations and, being 1 or 2 bits wide each, can easily be assembled into memories tailored to unusual bit-widths typical for tag-RAMs, thus saving on chip area. For compatibility with Linux, which generally uses pages of 4 kilobytes, a 32 bit virtual address is composed of 14 bits tag, 6 bits direct-mapped TLB index and 12 bits page offset. Hence, seven 2-bit RAMs with 64 entries each (**RAM64X2S** in Figure 4.19) are required for the tag. The physical address, on the other hand, only needs 16 of theoretically 20 bits for the page address since the stock ML310 only has 256 MiB of physical memory. These 16 bits fit in eight **RAM64X2S** blocks, yielding a total of 15 blocks as depicted in the figure. The tag comparator itself is realised using the fast carry-chain logic of the CLBs.

Beyond the HA-TLB, an associated FSM is able to walk the *GPP MMU-managed* page tables stored in main memory, consisting of the Page Global Directory and the Page Tables themselves. This FSM is responsible for performing a virtual-physical address

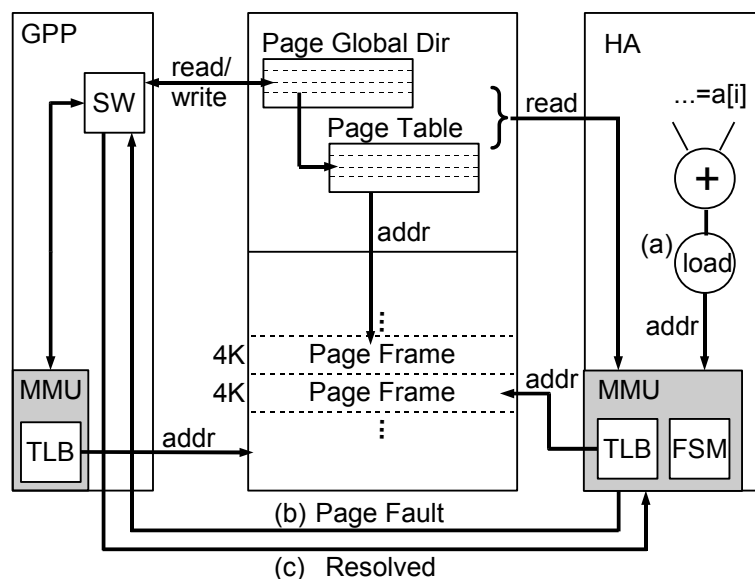


Figure 4.18.: PHASE/V TLB system: HA \leftrightarrow OS interactions

translation in case an HA-initiated memory access (here generated by the **load** node of its datapath, cf. Fig. 4.18a) leads to an HA-TLB miss. Should the mapping be already present in the HA-TLB, the translation only takes a single clock cycle, providing the HA with maximum throughput at minimum latency (**HAMEM**).

In addition to performing virtual-physical address translations, the PHASE/V scheme must also be able to handle page faults. These occur when the HA requests a virtual address that does not yet have (or no longer has) an associated page in physical memory. This condition will be detected when a page table walk performed by the HA-MMU cannot find a valid mapping. For its resolution, PHASE/V relies on the standard OS mechanisms: The FastPath signaling scheme introduced in Section 4.4 is used to request the handling of the page fault as a SW Service (shown as signal **Page Fault** in Fig. 4.18b). The Linux kernel then fetches the missing page frame, updates the page tables (signal **read/write**), and switches back to the HA to continue processing (signal **resolved**, Fig. 4.18c).

To handle the case when the OS flushes page frames from memory, or swaps them out to disk, the `flush_tlb_page` function of the kernel (the only function involved with flushing/swapping due to lack of free memory; recent kernels provide memory management notifiers [Corb08] as an official API instead) has been modified to not only invalidate the GPP-TLB, but also the HA-TLB, which is visible to the GPP in a memory-mapped fashion. Relying only on sniffing for GPP TLB- or page table writes would be insufficient, since not all HA-accessed pages will have been mapped into the kernel page tables before starting the HA.

Employing the standard OS mechanisms for page fault resolution also ensures that the HA can only access pages which are accessible to its associated SW process (with the correct owner/ group and access rights, achieving **PROTSYS**). Similar to AISLE,

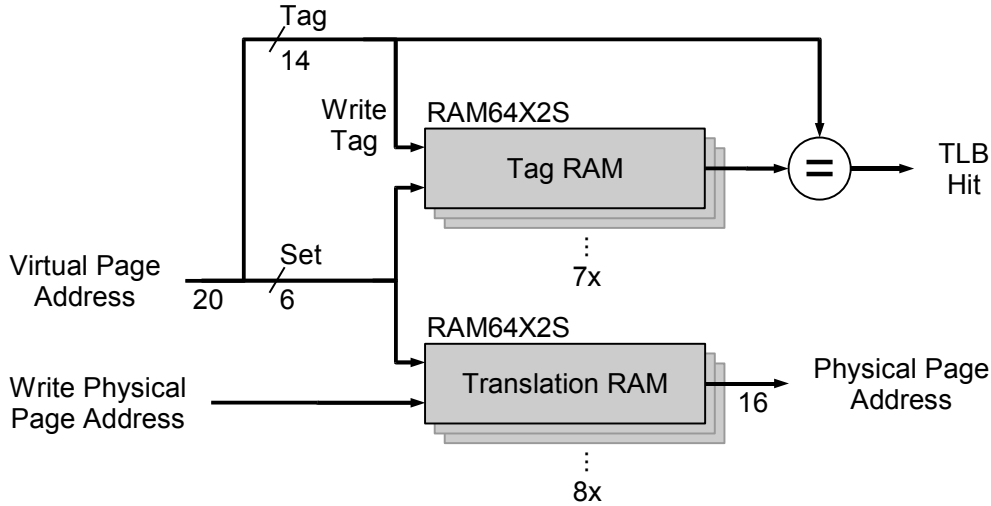


Figure 4.19.: PHASE/V TLB system: Tag- and translation RAM

PHASE/V also uses software to ensure cache coherency between GPP and HA. PHASE/V does support multiple HAs, either sharing the same HA-MMU or having dedicated MMUs (which would require explicit inter-HA-MMU coherency mechanisms).

4.6.3.1. Limitations

With PHASE/V, the HA now has the same capabilities as the GPP for virtual memory management. However, as will be examined in Section 8.1, these features are more costly than the simpler AISLE approach, both in terms of area and performance. It requires more HW (≈ 300 FPGA slices, see Table 8.1 in Section 8.1.2) and is also susceptible to the same performance risks as a GPP MMU: Whenever the working set of virtual addresses used by the HA exceeds the capacity of the HA-TLB, substantial address resolution overhead is induced by page table walks in rapid succession (also known as *TLB thrashing*). Although the problem can be addressed with a larger TLB, this in turn comes at the cost of larger chip area and eventually failure of single-cycle translation due to degraded timing. The alternative, a fully associative TLB would be more area efficient, but subject to even larger delays. Thus, only a limited set of addresses can be covered efficiently at full execution speed.

4.7. Chapter Summary

This chapter introduced a novel execution model as a fundamental basis of an efficient, high-performance Reconfigurable Computing Platform that this work aims to define. To this end, several existing execution models and their limitations were evaluated, and the flexible as well as universal Comrade execution model was found most suitable for achieving high computing performance. Further investigation identified the platform requirements which are imposed by such a powerful execution model. A commercial target

4. *Execution Model*

platform capable of meeting these requirements was subsequently analysed. Multiple issues in the vendor-supplied design concerning memory throughput and low-latency communication were identified and addressed. It proved advantageous to establish an independent channel for low-latency communication which bypasses both the slow standard hardware as well as operating system mechanisms. Furthermore, the memory attachment for the hardware accelerator was substantially improved without interfering with the rest of the system, especially the software running on the general purpose processor, while saving on resources by sharing logic with the existing memory controller. On the software side, these measures were complemented by three increasingly capable implementations for sharing identical pointers between hardware and software, culminating in a full virtual memory integration of the hardware accelerator. Thus, a simple yet powerful standard interface and its efficient implementation were provided, which tether well with the existing MARC caching and streaming system as well as the speculative memory system, platform management, and C-to-hardware interface, which form the other parts of this thesis and collectively constitute a platform target for automatic hardware/software compilers.

5. Platform Management

What is platform management? Why is it important for automatic HW/SW compilers in particular?

The heterogeneous and changing design environment requires an abstract and flexible representation of parameters. CoMAP, the Configuration Manager for Abstract Parameterizations, describes configurations from different architectures or tools in a single concept. It is a universal methodology for parameterizing a design (including interface description), its verification environment and constraints as well as the reference model. This abstraction bridges incompatibilities between different simulation, verification, synthesis, and cross translation tools. In particular, issues with unused code, tool-dependent HDL feature support, different implementations of multiple instance generation in Verilog and VHDL, and incoherent inter-HDL as well as SW/HDL parameterization features are addressed. Although the methods of CoMAP were originally developed for IP configuration and adaptation, they can also be applied to the integration of IP into (r)SoC as well as to building platform-based or derivative designs.

More important, they provide a precise Reconfigurable Computing Platform (RCP) *platform specification* for automatic HW/SW compilation, which can be used as well-defined and at the same time configurable target architecture by advanced HW/SW compilers such as Comrade [GäKo08] (see Section 6.5.1 in the next chapter). This platform specification consists of a set of hardware blocks, a formal description of their behaviours and physical interfaces as well as the interconnecting buses. Apart from a powerful execution model (see Chapter 4) to orchestrate the SW/HA processing flow and communication, Comrade uses these hardware blocks to construct the HA datapath. The complexity of such blocks ranges from very simple logic or arithmetic functions (e.g., logic AND, addition) to sophisticated hand-crafted intellectual property (IP) cores (the integration of the latter via Comrade will be shown in the next chapter). Thus, Comrade benefits from highly optimized complex HW functions while at the same time maintaining the flexibility of automatic datapath generation. To this end, all blocks including their interfaces are extensively parameterized, which enables Comrade to adapt both their functionality and interfaces to the needs of the automatically generated datapath. Nevertheless, they can also be manually adjusted to compose any arbitrary hand-crafted platform design.

The parameterization technique applied is two-fold: *Qualitative* parameterization addresses the in- or exclusion of code, thus choosing from architectural or functional alternatives. On the other hand, *quantitative* parameterization means the setting of values for attributes, e.g., bus widths or number of pipeline stages. Although being language independent, CoMAP is mainly targeted at hardware designs, since interface

properties and handshake protocols cannot be applied reasonably in a pure software environment.

CoMAP is a framework for hardware block configuration management in conjunction with interface property and protocol description. It consists of a repository and a set of related tools. The repository holds parameter sets representing block configuration data, relations which define parameter interdependencies, and static interface properties such as width, address type, and handshake protocol. Platform definitions group blocks to form a system resource that is accessible to other blocks, thus enabling interconnection and design reuse. All repository data is represented in an extensible, human-legible notation.

The main tool, which was implemented and used in commercial designs at an industrial partner, is the Hierarchical Preprocessor CoMAP-VPP (CoMAP-Verilog PreProcessor), which builds an instance of a parameterized design by stripping parameters and replacing them with their constant values. Thus, unintended results from Electronic Design Automation (EDA) tools (e.g., code coverage analysis) are avoided, which may occur if parts of code (e.g., optional functions, test logic) are disabled by regular program statements that evaluate parameterized expressions. Among the remaining tools is a parameter consistency checker, which evaluates the dependency relations, indicates inconsistencies, and proposes valid sets of values for the parameter concerned. The address finder supports the designer in address mapping and translation across bus bridges. A tool for automatic parameter extraction of different file formats helps the designer with building an initial parameter configuration.

In the following sections, CoMAP's data model and notation will be introduced, which are the fundamentals for the subsequent detailed descriptions of parameters, interfaces, components, and platforms.

5.1. Data Model

This section gives an overview of CoMAP's basic logical elements and their interplay. Descending the element hierarchy, these are *platforms*, *components*, *interfaces* and *parameters*. The hierarchical structure of the data model is shown in Figure 5.1.

A platform usually consists of one or more *configurations*. Each platform configuration instantiates several components that are linked to blocks (IP cores, bridges, ...), which in

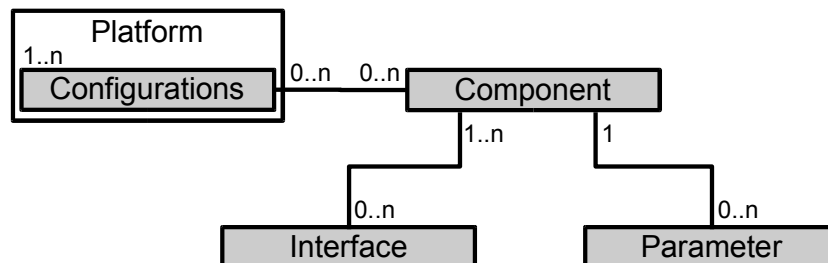


Figure 5.1.: CoMAP data model

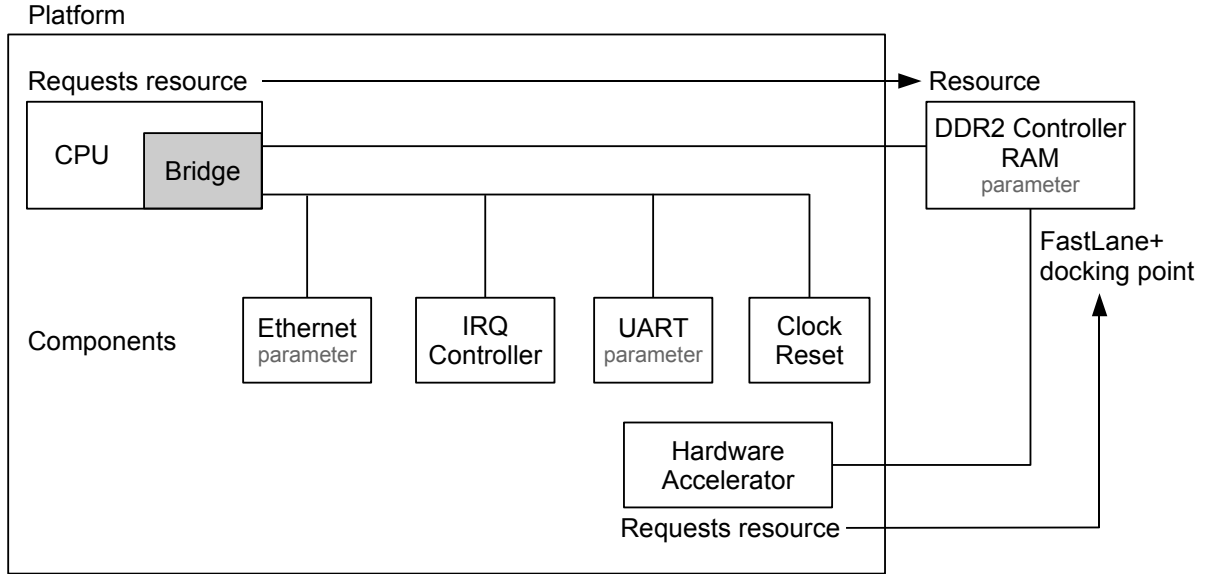


Figure 5.2.: Simplified Xilinx ML507 platform represented by CoMAP elements

turn feature interfaces and parameters. Interface descriptions may be reused as *templates*, whereas a parameter is local to its component. Components are not necessarily part of a platform, e.g., IP cores exist independently of a platform. Moreover, platforms may advertise a set of *resources*, which are docking points for user logic that offer a certain functionality. The user logic (including IP cores and even software via GPPs and PaCIFIC, see Chapter 6) can in turn request a resource from a platform. Since the actual connection points between user logic and resource are interfaces, resources are managed within the interface description and binding process, which will be described in Section 5.4.1.

A simplified version of a real rSoC (Xilinx ML507 [Xili09c]) (shown in Figure 5.2) consists of a DDR2 SDRAM controller attached to RAM, an interrupt controller, and Ethernet as well as UART interfaces. The embedded CPU is connected to memory and peripherals via an internal bus bridge. Furthermore, a hardware accelerator acts as user logic. In the CoMAP data model, the HA as well as all other rSoC elements are *components* which optionally carry *parameters*. The *platform* comprising the rSoC without the DDR2 controller advertises a *resource* via a dedicated *interface* docking point (the FastLane+ interface in Figure 5.2). Components such as the CPU can request *resources* (e.g., the DDR2 SDRAM controller), thus implicitly adding them to the platform.

The *CoMAP repository* stores all configuration data for IP cores, their interfaces, platforms, and interconnection. The data model is build from a human-readable description language which supports structure by instantiation and inheritance. Its notation is presented in the next section.

5.2. Notation and Basic Grammar

As a short excursion from CoMAP’s functional description, this section introduces the notation and grammar fundamentals which are used throughout the remainder of this chapter to define the elements of the CoMAP repository. The notation is based upon a block-oriented language such as VHDL or Modula. Hence, the basic syntactical grouping element is a block. A block consists of three parts: the declaration, which is a line containing the keyword for the type of block, the body, and the end delimiter, which repeats the block type keyword prefixed by “**end**”. Keywords followed by a colon “:” signify a statement. There is no statement delimiter, multi-line statements are possible. In contrast to VHDL and Modula, parameters are passed by name exclusively. The standard parameter-passing scheme in round brackets () is replaced with the parameter name followed by its value. (e.g., **name: value**). This provides for an unambiguous assignment of parameter values as is known from Smalltalk-80 [GoRo83]. In the following descriptions, the notational conventions of the Extended Backus-Naur Form (EBNF) [IsoE96] were adapted as follows:

- Literals and terminal symbols (i.e., EBNF tokens) are printed in **bold monospaced font**.
- All other code is set in *slanted monospaced font*.
- **a / b** signifies either a or b.
- **[c]** is zero or one of c.
- **{d}** is one or more of d.
- **e..z** means the range from e to z.
- EBNF comments are delimited by **#** and the end of the line
- CoMAP comments start with **;** and also continue to the end of the line.
- Identifiers and keywords are case sensitive.

The grammar which is presented in this chapter is constructed and explained incrementally to avoid redundancies. Hence, all EBNF non-terminal symbols that are not described within a certain section can be found in an earlier section. For best comprehension, it is recommended to read this chapter sequentially.

The top level grammar of CoMAP shows that a file may contain an arbitrary number of platforms, components, and interface templates:

```
comap_file ::=
    comap version_number
    [{platform}]
    [{component}]
    [{interface_template}]
    end comap
```

Different from platforms and components, *interface templates* do not directly represent implemented logic objects. Instead, they provide abstract interface properties for inheritance and reuse by other interfaces (described in Section 5.4). The **version_number** following the **comap** keyword is a hint for tools (cf. Chapter 5.7) to choose the appropriate parser. Version numbers consist of a major and one or more minor parts separated by dots.

The basic grammar building blocks including expressions, numbers, and identifiers are shown below. They are often referred to from other parts of the grammar described later in this chapter:

```

expression ::=
    simple_expression [shift_operator simple_expression]

shift_operator ::=
    << | >>

simple_expression ::=
    [sign] term [{sign term}]

sign ::=
    + | -

term ::=
    primary [{multiplying_operator primary}]

multiplying_operator ::=
    * | / | mod

primary ::=
    identifier
    | integer
    | pos_float
    | string
    | discrete_range
    | function_call
    | ( expression )

identifier ::=
    letter [{[_] letter_or_digit}]

letter_or_digit ::=
    letter | dec_digit

letter ::=

```

5. Platform Management

```
A..Z | a..z

version_number ::=
    decimal [{. decimal}]

pos_float ::=
    decimal . decimal

integer ::=
    [-] natural

natural ::=
    hexadecimal | binary | octal | decimal

hexadecimal ::=
    0x {hex_digit}

binary ::=
    0b {bin_digit}

octal ::=
    0o {oct_digit}

decimal ::=
    {dec_digit}

bin_digit ::=
    0 | 1

oct_digit ::=
    bin_digit | 2 | 3 | 4 | 5 | 6 | 7

dec_digit ::=
    oct_digit | 8 | 9

hex_digit ::=
    dec_digit | a | b | c | d | e | f | A | B | C | D | E | F

string ::=
    " [{ASCII_7bit_printable}] "

discrete_range ::=
    expression .. expression
```

```

function_call ::=
    function (expression [{, expression}])

function ::=
    pow / log / abs

```

CoMAP expressions are similar to VHDL expressions with minor modifications. They are used in boundary functions and dependency relations which are both explained in the next chapter. Note that the expressions in a *discrete_range* are supposed to evaluate to integer values. The special function `pow(base, exponent)` raises *base* to the power of *exponent*. The logarithm of *x* to the base *base* is calculated by `log(base, x)`. The absolute value of *x* is returned by `abs(x)`.

Based on the fundamental grammar introduced in this section, the following sections present CoMAP's platform management objects, starting with parameters.

5.3. Parameters

There are several definitions for the term *parameter*, and at least as many notions of a parameterization. In CoMAP, parameters carry configuration data for the hardware block (IP core, bus bridge, controller etc.), which is associated with a component (see Section 5.1). Hence, parameters are the main vehicle for manipulating the functionality of a component, both quantitatively (e.g., bus widths, number of pipeline stages) and qualitatively (e.g., inclusion of optional functions).

Parameterizable file formats provide many different kinds of parameter constructs. However, there are two distinguishable parameter categories. First, there are non-hierarchical parameters. They are valid only in the same module, block, or section where they are defined, and are supported by all parameterizable file formats (e.g., VHDL `constant`). The second kind is the hierarchical parameter. It is valid in the same block where it is declared and may be initialized with a default value, but generally this value is overridden by a definition on a higher hierarchical level (e.g., VHDL `generic/generic map`, Verilog `parameter/defparam`). Most parameterizable file formats support hierarchical parameters (shown in Figure 5.3). These two types of parameters are sufficient to cover the parameterization process of a design, while the restriction to only two classes enables the migration into a single notation. Table 5.1 maps the representatives of both classes for selected file formats.

Reflecting both qualitative/quantitative as well as hierarchical parameterization techniques, a *parameter* is an abstract data type which is characterized by the properties shown in Table 5.2. *Name* and *location* uniquely identify the parameter, *value* and *type* define its payload. The processibility of a parameter is controlled by its *modifiable* and *strippable* flags, while *dependants* and *boundary function* define its relation to other parameters, thus providing intrinsic support for definition and checking of parameter interdependencies. These properties will be explained in detail in the context of the formal parameter definition, which is described by the following grammar:

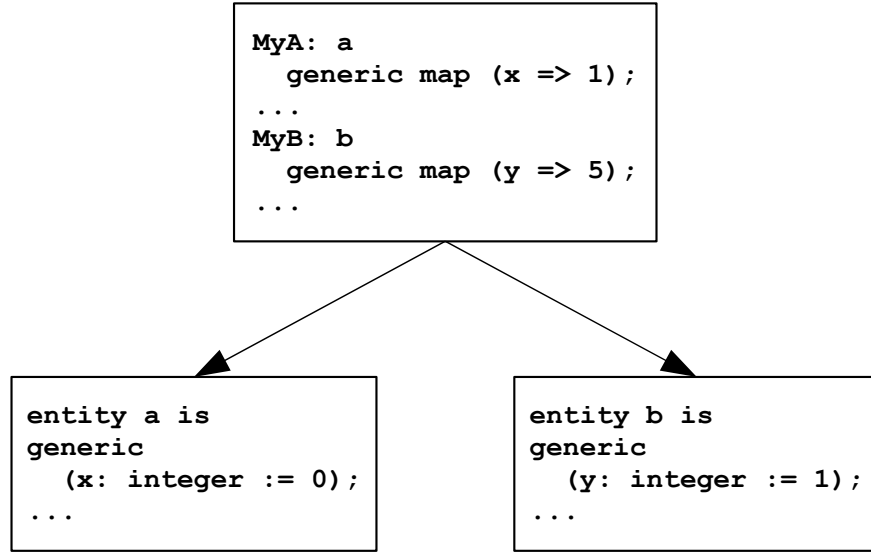


Figure 5.3.: Hierarchical parameters in VHDL [LaRa02]

	Hierarchical Parameter	Non-hierarchical Parameter
VHDL	generic/generic map	constant, generic
Verilog	parameter/defparam	parameter, 'define
C	-	constant, #define
SystemC	Template class Constructor parameter	constant, #define
Specman e	Hierarchical (generation) constraint	Flat (generation) constraint
Constraint file	Placement constraint	Timing/placement constraint
Synthesis script	-	File path, option variable

Table 5.1.: Parameter classes in selected file formats

Property	Description
name	Name, unique in location
location	Location in the design hierarchy
comment	Informal parameter description
value	The current parameter value
type	Data type
modifiable	Parameter may be modified
strippable	Parameter may be replaced by its literal value
dependants	Other parameters depending on this one
boundary function	Function delimiting legal values

Table 5.2.: Parameter properties

```

parameter ::=
    parameter parameter_name
    location
    [comment]
    parameter_value
    parameter_type
    modifiable
    strippable
    [dependants]
    [boundary_function]
end parameter

```

```

parameter_name ::=
    identifier

```

The parameter definition is a syntactic block which starts with the keyword **parameter** and an identifier which represents its name. Since a parameter is visible only inside the component where it is defined and in all interface templates that are instantiated from that component, the name space for parameters is limited to components.

```

location ::=
    location: string

```

The **location:** statement expects a string that contains either the position of the parameter in the design hierarchy, or the physical location in a file. The hierarchical position is given in a form of **x/y/z**, where **x** represents the top level block of the design, **y** is a block in the path down the hierarchy, and **z** marks the block where the parameter is defined. The scope and granularity of the blocks depend on the type of HDL that is used for the design. The physical location is given by **filename:line:column**, **line** and **column** counts start at zero. A unique key to the parameter is formed by its location and its name (identifier).

The optional comment block, which follows the location, may be used to convey additional information. In contrast to standard CoMAP comments (introduced by **;**), the comment blocks are logically linked to their superordinate block, thus providing localized informal data:

```

comment ::=
    comment
    [{string}]
end comment

```

The **value:** statement carries the value of the parameter as a string:

```

parameter_value ::=
    value: string

```

The **type:** clause gives a hint on how the parameter value has to be interpreted:

```

parameter_type ::=

```

```
type: parameter_type_specifier
```

```
parameter_type_specifier ::=
    string / integer / float / boolean / enum
```

The types **string**, **integer**, and **float** are self-explanatory, their ranges resp. accuracies depend on implementation. While **boolean** can assume the values **true** and **false**, **enum** carries multiple values as a list: `{"stringval", 1, 3.14, false, 0}`. The base types of the list are the four types just mentioned. Parameters of type **enum** must not have any dependants (see below). If the parameter value may be changed in the repository, set **modifiable:** to **yes**:

```
modifiable ::=
    modifiable: yes_or_no
```

```
yes_or_no ::=
    yes / no
```

If a parameter stripper or preprocessor (described later in Section 5.7) is supposed to replace the parameter in the design by its actual value, set **strippable:** to **yes**:

```
strippable ::=
    strippable: yes_or_no
```

The **dependants** block defines a list of parameters which depend on the **value:** of the current parameter. Every time the current parameter is changed, all members of this list are triggered to check and, if need be, recalculate their values based on their respective **boundary** function (specified below), or alternatively all dependency relations (part of a component declaration, see Section 5.5.1) which they are part of. Parameters on the list are identified by their unique key **name:** and **location:**. This key either has to be provided manually by the HW designer or it is automatically generated by the Parameter Extractor (described later in Section 5.7). The designer is supported by the Parameter Consistency Checker (also described in Section 5.7) in determining the dependants of each parameter. Note that a parameter can reference itself on its dependants list, e.g., to trigger a self-check of its value on every update:

```
dependants ::=
    dependants
    {dependants_body}
end dependants
```

```
dependants_body ::=
    name: parameter_name
    location
```

An example parameter definition is shown in Figure 5.4. Two polynomial or special (e.g., **log**, **pow**, **<<**; cf. Section 5.2) **boundary** functions mark the upper (**max:**) and lower (**min:**) bounds of a continuous interval, respectively. The actual parameter value must be an element of this interval to be considered as valid:


```

parameter data_width
  location: "marc/cache_port"
  value: "32"
  type: integer
  modifiable: no
  strippable: yes
  dependants
    name: tag_width
    location: "marc/tagram"
    name: data_width
    location: "marc/cache_port"
  end dependants
  boundary
    max: 64
    min: 8
  end boundary
end parameter

```

Figure 5.4.: CoMAP parameter definition

```

boundary_function ::=
  boundary
  boundary_max_min | boundary_is
end boundary

boundary_max_min ::=
  max: expression
  min: expression

boundary_is ::=
  is: expression

```

These functions may depend on other parameters, constants or complex expressions. However, ranges are not allowed as primary arguments. If the **max:** and **min:** expressions are identical, a single function defined by **is:** may be used instead. Boundary functions are local to a parameter and must not induce interdependencies with other parameters. Instead, complex interdependencies are expressed using *dependency relations*, which describe sets of discrete values that a parameter may assume. Since dependency relations define interdependencies for *multiple* parameters, they are hosted in the logically superordinate *component* definition (specified in Section 5.5).

Although the versatile parameter definition is designed to accomodate all sorts of diverse and complex configuration data, including its location in a design hierarchy, parameters do not convey logical grouping, which is however required by most modern

design methods. Hence, CoMAP employs platforms and components for modularization (described later), the latter as well to logically combine parameters. Likewise, interfaces are used to logically group interconnection resources, which will be described in the next section.

5.4. Interfaces

Interfaces describe the connectivity and protocol interactions of HW blocks based on a set of I/O ports (described later in Section 5.4.2) and their interplay. A block can be an IP core, a bus bridge, or a resource such as RAM or an I/O controller (network, SATA, USB, etc). The interface specification (summarized in Table 5.3) provides information about the type of the interface, its associated ports, the traffic scheme, the associated address port (if any), the address mode (none, multiplexed, ...), the data structure of the used protocol, the respective handshake signals, and the kind of arbitration. An interface may be defined directly in a component definition (specified later in Section 5.5), or deposited as a *template* for later (re)use. It acts as initiator or target in the *interface binding process*, which will be described in Section 5.4.1. The grammar for an *interface description* is identical to that of a template and is shown below:

```

interface ::=
    interface interface_name
    interface_body
end interface

interface_template ::=
    interface

interface_name ::=
    identifier

interface_body ::=
    [template_instantiation]
    [docking_mode]
    interface_type
    [interface_subtype]
    header
    {port}

template_instantiation ::=
    template: interface_require

interface_require ::=
    interface_name [require: [exact] version_number]

```

Property	Description
name	Name, key for interface binding
author/organization	Originator information
comment	Informal comment
template	Inherit all body elements from this interface template
docking mode	Interface requests or is external resource
type	Type, unique key for binding with subtype and version
subtype	Subtype, unique key for binding
version	Version number, unique key for binding
port	Port(s) which this interface comprises

Table 5.3.: Interface properties

```
interface_type ::=
    type: identifier / custom
```

```
interface_subtype ::=
    subtype: identifier
```

An interface definition, which is a block statement, contains an optional template instantiation which can be used to group and reuse the same or similar interfaces in a fashion analogous to the classes and inheritance of object-oriented programming.

```
interface simple
    template: simple_template require: 2.1
    ...
end interface
```

Figure 5.5.: CoMAP interface definition using a template

In the example that is shown in Figure 5.5, the interface *simple* inherits from *simple_template*. All body elements from the template are adopted unless they are overridden by a local redefinition. The template may also instantiate another template, whereas recursion must be avoided. The optional **require: [exact]** extension of the **template:** statement claims at least (or exactly, if **exact** is given) version 2.1 of *simple_template*.

The following sections describe the mechanism for connecting interfaces (*interface binding*), and ports as their basic elements.

5.4.1. Interface Binding

When building platforms or designs, interfaces must be connected. CoMAP provides heuristics which automatically compose designs by connecting interfaces with matching characteristics. The interface binding process is started when the platform or design is

composed for later use. It iterates over every unconnected interface, selects it as initiator, i.e., the one that is seeking a matching peer, and connects it subsequently to a target interface until all interfaces are bound.

Multiple bindings per target interface enable the construction of *buses*. As usual, buses are used in CoMAP to model arbitrary $n:m$ connections between multiple interfaces. To automatically generate arbitration logic, all peers on a bus must use the **access** elements of the ports involved (see Section 5.4.2) to identify their roles as bus masters or slaves. All slaves share the same input signals from the active master via the arbiter, while all slave outputs are logically ORed and subsequently presented to the bus master(s). Thus, an implicit distributed multiplexer is automatically realized among the slave outputs. Different multiplexer types can optionally be implemented as dedicated external logic.

Special care is taken for interfaces that either request or offer a *resource* (also described in Section 5.1). Such interfaces are optionally used for connecting a component to additional building blocks (e.g., RAM, I/O, HW accelerators) that a platform provides. In contrast to buses, resources provide $1:n$ connections, where a single resource can be shared by multiple components (but not vice versa). Figure 5.6 shows the stepwise binding process and the conditions (enumerated as **a** to **d** in the figure) under which a potential target interface matches an initiator.

Condition **c** provides a match by interface type. The interface **version:** tag (part of the **header** grammar construct shown below) is associated with **type:** and **subtype:** (see grammar in the previous section). All three form the unique key for interface binding. When used in a template, this version number is referred to by the template instantiation statement. Version numbers consist of a major and one or more minor parts separated by dots. The mandatory interface type (**type:**) can be an arbitrary identifier, which typically denotes standard interfaces or buses (e.g., AMBA AXI, CoreConnect PLB/OPB, i960), or the predefined identifier **custom**. The latter is used when no particular type can be determined for the interface. The optional body element **subtype:** is an identifier that distinguishes sub-categories for the generic interface type, e.g., different parts or operating modes of a complex IP core interface. Well-known interface types and subtypes should be defined in a separate interface library specification (not part of this work).

Conditions **a** and **d** identify the valid docking points of a platform (described later in Section 5.6) for user logic, IP cores, and software. An interface which requests a platform resource via a *docking_mode_specifier* (find grammar below) that equals **exclusive** or **shared** always acts as initiator in the binding process, it can never assume the target role. Otherwise, two or more component interfaces which request identical resources would be compatible and hence illegally connected to each other. Furthermore, unconnected component interfaces that request a resource can safely be distinguished from the resource interfaces themselves, which have to be explicitly labelled as **resource** by means of the *docking_mode_specifier*, and must not be initiators. Otherwise, multiple identical resource interfaces would also be compatible and thus illegally connected.

However, the resource selection can also be influenced directly: If the platform advertises a resource under a well-known interface name, such dedicated resource (e.g., a certain I/O device) can be precisely selected via interface names (condition **b**).

If an initiator interface has the docking mode specifier **shared**, it connects to a target

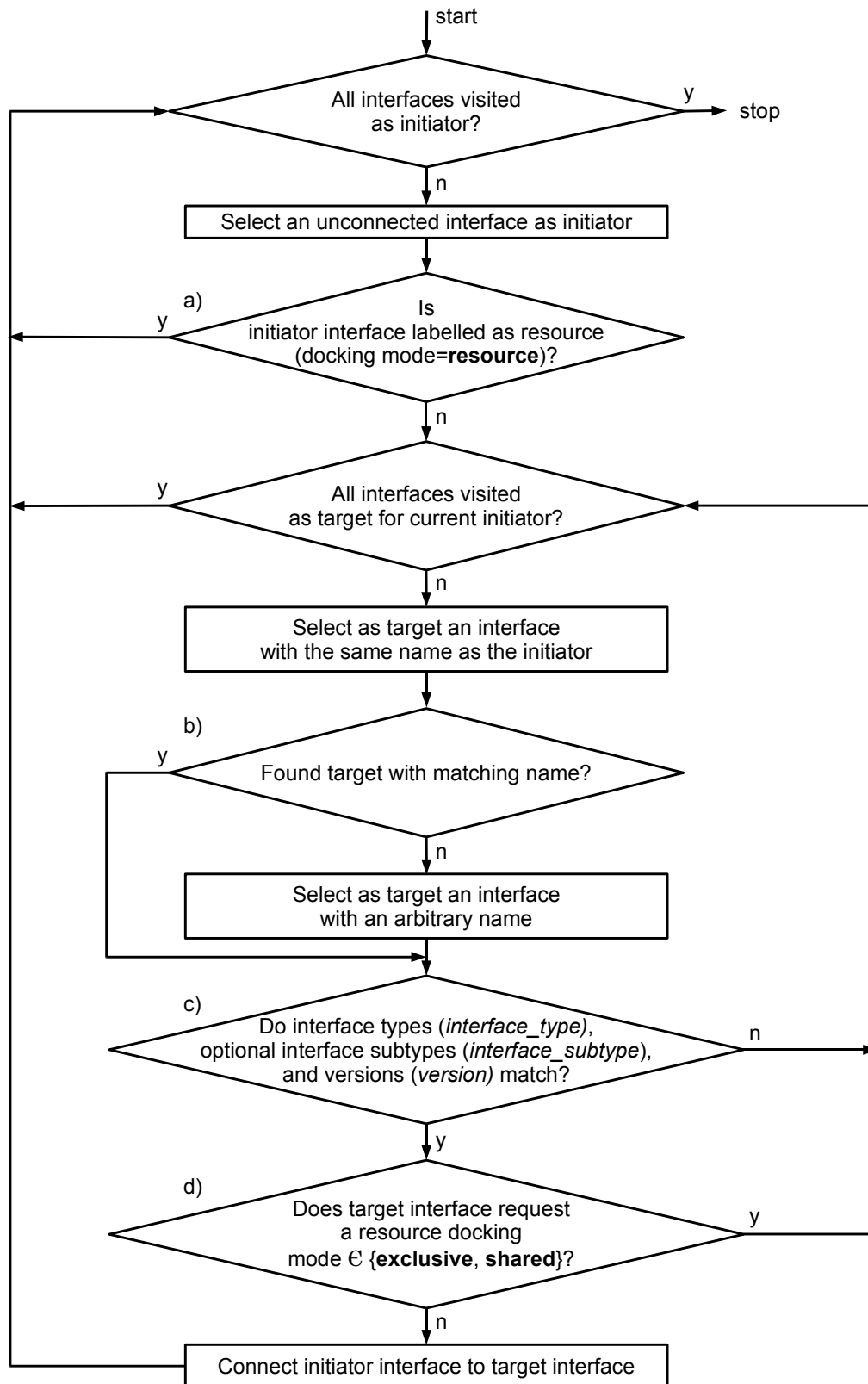


Figure 5.6.: Interface binding process

interface even if other interfaces are already connected. Hence, arbitration logic is required which is generated from the **access** elements of the ports involved (see Section 5.4.2). By contrast, an initiator interface with docking mode **exclusive** requests a peer which is exclusively available for the initiator without any arbiter. The optional **persistent** tag demands a target interface which provides the same internal state and data after the initiator block has been disabled or removed (e.g., during reconfiguration or low-power sleep) and re-enabled again.

The example that is shown in Figure 5.7 presents two interfaces (both named **zbt_ram**) which are bound by CoMAP and connected port by port. The initiator in the interface binding process (**zbt_ram**) requests an exclusive resource of type **zbt**, subtype **flowthrough**, and version **1.0**. The resource interface **zbt_ram** matches this requirement as target and hence is bound to the initiator. The latter interface cannot be the initiator due to its **resource** label. Hence, it also cannot connect to **zbt2** and vice versa. The initiator would be compatible with **zbt2** as well, but precedence is taken for **zbt_ram** due to the matching names.

```

; Initiator:
interface zbt_ram
    exclusive
    type: zbt
    subtype: flowthrough
    version: 1.0
    ...
end interface
; Target:
interface zbt_ram
    resource
    type: zbt
    subtype: flowthrough
    version: 1.0
    ...
end interface
; Other resource:
interface zbt2
    resource
    type: zbt
    subtype: flowthrough
    version: 1.0
    ...
end interface

```

Figure 5.7.: CoMAP interface binding

Below find the *header* and *docking_mode* grammars:

```

header ::=
    version
    [author]
    [organization]
    comment

author ::=
    author: string

organization ::=
    organization: string

version ::=
    version: version_number

```

The optional **author:** part of the *header* construct is a string that should be treated informatively and contains the author's name who provided the code. Also informatively, **organization:** and *comment* (see Section 5.3) should be used to provide additional information.

```

docking_mode ::=
    [persistent] docking_mode_specifier

docking_mode_specifier ::=
    shared / exclusive / resource

```

Having established in this section the rules and mechanism for automatic interface connection, the next section will present ports as the core element of interfaces.

5.4.2. Ports

Ports play a central role in interface definitions. They represent external connections of a hardware block, their attributes, and simple interface protocols. Furthermore, they provide information on how to generate arbitration logic if necessary, and data traffic characteristics to adapt caching or streaming strategies. The properties of a port are shown in Table 5.4. A port is uniquely identified within an interface definition via its *name*. Hence, the same port name may be reused in different interfaces. The physical characteristics of a port are described by its *direction* (e.g., I/O), and *width*. Protocol semantics which control communication with other ports or interfaces are defined by *transaction*, *sequence*, *handshake*, and *access*, while the cooperation of ports in the same interface is controlled by *clock* and *address*. Finally, *traffic* describes the statistical traffic pattern for this port, which can be exploited to generate an optimally matched memory hierarchy (e.g., a speculative memory system, see example in Section 7.5), and for rate matching between interfaces. The **port** statement is block-based as usual with declaration, body, and end delimiter.

5. Platform Management

Property	Description
name	Name, local to interface, connect by name
comment	Informal comment
transaction	type of protocol semantics
direction	input, output, or both
width	port width in bits
clock	name of reference clock
address	name of associated address port
sequence	data type and static protocol
handshake	handshake signal(s) for sequence
access	type of bus arbitration
traffic	data traffic characteristics

Table 5.4.: Port properties

```

port ::=
    port port_name
    [comment]
    transaction
    direction
    width
    [clock]
    [address]
    [{sequence}]
    [handshake] [handshake]
    [access]
    [traffic]
end port

```

```

port_name ::=
    identifier

```

After the `port_name` and an optional comment block, the **transaction:** statement determines the basic type and semantics of the port:

```

transaction ::=
    transaction: transaction_body

transaction_body ::=
    data / interrupt / transaction_clock / transaction_reset

transaction_clock ::=
    clock min: speed_value max: speed_value

transaction_reset ::=

```



```

    reset_type natural cycles

reset_type ::=
    negreset / posreset

speed_value ::=
    pos_float [multiplier]Hz

multiplier ::=
    p / n / u / m / k / Ki / M / Mi / G / Gi / T / Ti

```

The value **transaction: data** means that the port is used for standard data transfer and the data semantics for *sequence* and *handshake* are selected. The behaviors of *sequence* and *handshake* for certain semantics are described later in this chapter. The value **interrupt** selects the interrupt behavior (also described later), **clock** designates the port as a clock (shown in Figure 5.8):

```

    port clk
        transaction: clock min: 25 MHz max: 30 MHz
        direction: input
        ...
    end port

```

Figure 5.8.: CoMAP clock port

The speed value in MHz defined by **min:** indicates the minimum clock speed at which the interface or block (depending on implementation) can operate, **max:** is the maximum speed. The *multiplier* indicates the common SI/IEC prefixes as powers of ten (pico, nano, micro, milli, kilo, Mega, Giga, Tera) or powers of two (Kibi, Mebi, Gibi, Tebi). Note that an interface (see Section 5.4 above) may have multiple clock ports.

The mandatory **direction:** statement marks the port as **input**, **output**, or **inout** (bidirectional), which corresponds to the port directions known from HDLs:

```

direction ::=
    direction: direction_body

direction_body ::=
    input / output / inout

```

The **negreset/posreset** transaction types (shown in Figure 5.9) declare the port as a reset. Both are followed by the minimum number of clock cycles that the reset must be consecutively active to operate correctly. The value **negreset** is active low, **posreset** is active high. Note that the reset behavior is application-specific, e.g., it could merely apply to the interface that the port is part of, or cause the reinitialization of the whole component.

The obligatory **width:** clause determines the vector size of the port in bits:

```

port reset
  transaction: negreset 50 cycles
  direction: input
  width: 1
end port

```

Figure 5.9.: CoMAP reset port

```

width ::=
  width: width_specifier

```

```

width_specifier ::=
  natural | identifier

```

The remaining body elements of a port (see beginning of this section) are optional. The **clock:** element links the port to a clock port named *port_name*. The clock port must be defined in the same interface definition. All signal changes on the port are expected to occur synchronously to the active edge of this clock, which is defined by **polarity:**. If no clock is defined, transfers over the port are asynchronous:

```

clock ::=
  clock: port_name polarity: polarity_specifier

```

```

polarity_specifier ::=
  pos | neg | both

```

Apart from associating a port with a clock signal, pairs of data and address can be defined as well. The **address** element either binds the port, which in this case is regarded as a *data* port, to a dedicated address port (**port:**), or declares the port as temporally **multiplexed:** data *and* address port. In both cases, **logical:** determines which part or range of an address is logically visible to the port. Thus, an implicit translation is established between external (**physical:** or **multiplexed:**) and internal (**logical:**) address representations. The grammar for the **address** element will be illustrated below with two examples:

```

address ::=
  address address_type logical: bus_part_or_range

```

```

address_type ::=
  multiplex_specifier | address_port

```

```

multiplex_specifier ::=
  multiplexed: bus_part_or_range

```

```

address_port ::=

```

```

port: port_name physical: bus_part_or_range

bus_part_or_range ::=
    bus_part / discrete_range

bus_part ::=
    expression : expression

    port data
      transaction: data
      direction: output
      width: 32
      address port: addr physical: 21:0 logical: 23:2
      ...
    end port

```

Figure 5.10.: Linking a data port to an address port

In the example which is shown in Figure 5.10, the port **data** is associated with the address port **addr**. Assuming that **addr** is an input, the physical bits 21 to 0 of port **addr** are interpreted as a logical address from bits 23 to 2, i.e. **addr** is shifted left by two and right-padded with zeros for the logical representation (shown in Figure 5.11). If **multiplexed:** is used instead of **port:**, both address and data are transferred alternately over the same port (address first), while the address arithmetic just mentioned still applies. Note that instead of bus parts, ranges may be used as well (cf. Figure 5.12).

The second example (shown in Figure 5.12) demonstrates an address translation as is common in bus bridges (see Figure 5.13). Here, the configuration of the address translation is fixed inside the bus bridge. However, an arbitrarily programmable address translation can be modeled by using parameters for the ranges. Assume that a producer

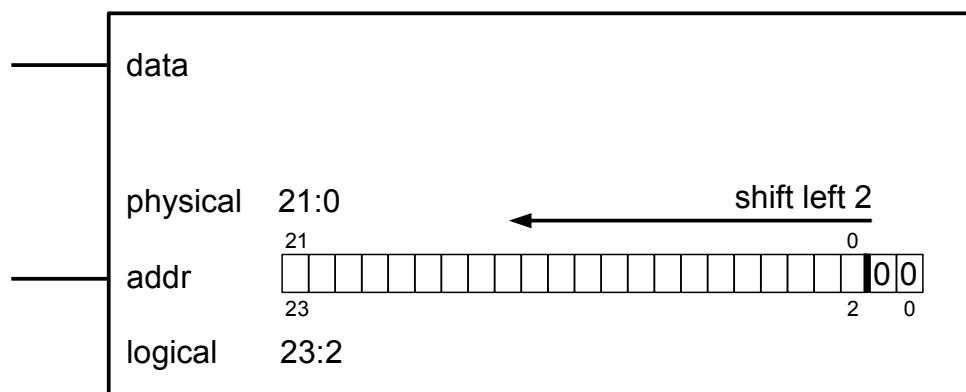


Figure 5.11.: Physical and logical address representations

```

port opb
  transaction: data
  direction: input
  width: 32
  address port: addr_opb
    physical: 0x1F40..0x1F80
    logical: 0x0..0x40
  ...
end port

port pci
  transaction: data
  direction: output
  width: 32
  address multiplexed: 0x8000..0x8040
    logical: 0x0..0x40
  ...
end port

```

Figure 5.12.: Address translation using ranges

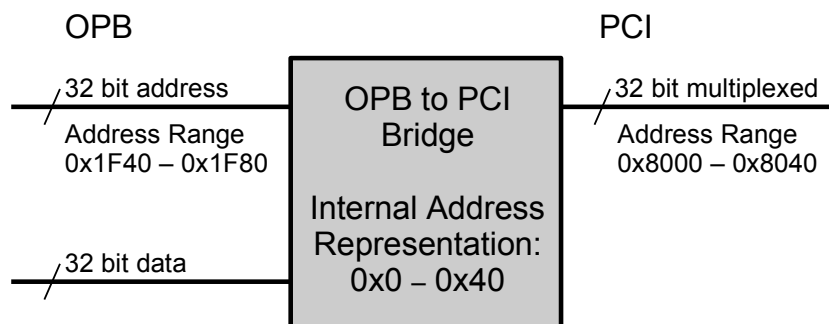


Figure 5.13.: A bus bridge translates address ranges

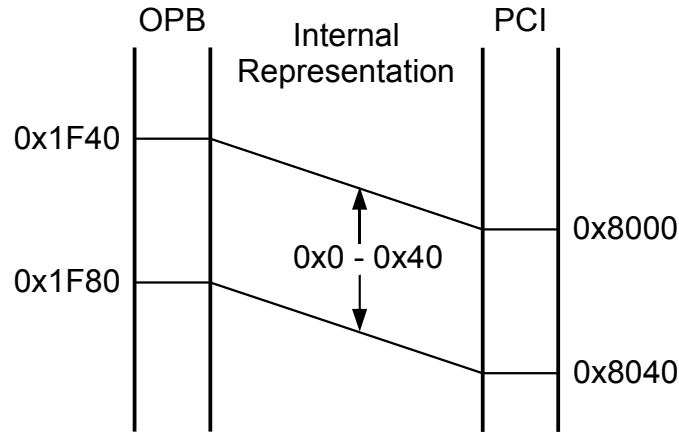


Figure 5.14.: Address ranges are translated via an internal representation

on the OPB bus is writing to a consumer on the PCI bus. On the OPB bus, the addresses ranging from 0x1F40 to 0x1F80 are translated to a bridge-internal representation from 0x0 to 0x40 (shown in Figure 5.14). Then this internal representation is again translated to the PCI address range from 0x8000 to 0x8040.

Complementing the basic port features explained above, the next two sections present mechanisms to describe port protocols, and arbitration as well as data traffic characteristics.

5.4.2.1. Sequences and Handshakes

The static protocol of a port is controlled by optional *sequences* and at most two *handshakes*. A **sequence** associates a port with the data part of a port protocol and defines the nature, order, and count of the data items that are transferred over the port or bus. The control part of the protocol is provided by the *handshake* construct (described next). Both handshake and sequence declarations can be reused via interface templates (see Section 5.4 above), which they are part of. The sequence is also used for the static part of software interface generation with PaCIFIC (described in the next Chapter 6), which adds a dynamic protocol extension that associates a sequence with an Abstract Data Type (ADT). The properties of a sequence are summarized in Table 5.5. Each *endianness/bit/(un)signed* declaration defines a data item/ADT member. The sequence/ADT is optionally replicated *repeat* times. Note that only the protocol container or ADT is replicated, the actual data content may change on every iteration. With this

Property	Description
repeat	Total sequence repetitions, iterations are triggered by handshake
endianness ... bit	Defines one of multiple sequence items (byte order, width)
(un)signed	Item is in plain or two's complement representation

Table 5.5.: Sequence properties

mechanism, structured data arrays can be transferred contiguously over the port.

```
sequence ::=
    sequence [repeat: repeat_count]
    {sequence_body}
end sequence
```

```
sequence_body ::=
   endianness: natural bit signum
```

```
repeat_count ::=
    natural / inf
```

```
endianness ::=
    bigendian / littleendian
```

```
signum ::=
    unsigned / signed
```

The **sequence** keyword starts a block which may contain multiple *sequence_body* grammar constructs. The *sequence_body* usually consists of several lines, each stating the endianness of a single data item, its width in bits, and whether it can be negative (**signed**).

```
port abstract_data
    transaction: data
    direction: input
    width: 32
    sequence repeat: 5
        bigendian: 32 bit signed
        bigendian: 8 bit unsigned
    end sequence
    ...
end port
```

Figure 5.15.: A sequence defines the data protocol for an abstract data type

The data items are expected to be transferred over the port in the order that is defined by the sequence. In the example shown in Figure 5.15, the signed 32 bit data is transferred first, followed by the unsigned 8 bit data. An optional **repeat:** count defines how many times the data transfers defined by the sequence body are repeated. Every iteration of a sequence body, which transfers one instance of the entire ADT, is triggered by the handshake mechanism described below. An infinite repeat count (**repeat: inf**) indicates that the sequence repetition never terminates, and data is thus streamed continuously over the port. However, a component may choose to suspend

Property	Description
<code>type</code>	Direction and asserted state of handshake
<code>name</code>	Name of handshake port
<code>offset</code>	Offset between handshake and data in clock cycles
<code>latency</code>	Initial target handshake latency after initiator

Table 5.6.: Handshake properties

Direction Asserted State	Incoming	Outgoing
High	enablein	enableout
Low	stallin	stallout

Table 5.7.: Handshake types

the data transfer at any time via the handshake mechanism, e.g., depending on the transferred data content (a header which determines the size of the following payload). Multiple sequence definitions per port are processed sequentially.

handshake ::=

handshake_type name: port_name offset: integer latency: natural

handshake_type ::=

enableout / stallout / enablein / stallin

The handshaking scheme (an overview of its properties is shown in Table 5.6) consists of an incoming and an outgoing signal per port, which both are declared by *type* and *name*. *Offset* and *latency* define the temporal relationship of the handshakes that are involved in the port protocol. The **handshake** element (see grammar above) links the port to a handshake port which is indicated by **name:**. The following descriptions are valid for ports of type **transaction: data**. There are four types of handshakes: The first two named **enableout** and **stallout** declare the handshake port as an outgoing handshake, while the remaining two (**enablein**, **stallin**) declare an incoming handshake. For an outgoing signal, the asserted state, which is selectable as high (**enableout**) or low (**stallout**) means that a component is ready to consume data (on an input port) or that data is waiting to be fetched (on an output port). The incoming signal is connected to the outgoing signal of the port at the peer end of the communication. Table 5.7 summarizes the possible combinations.

It is not necessary to specify both incoming and outgoing signals, a one-way handshake is possible as well as no handshake. A transaction is considered complete when all specified handshake signals are simultaneously active at a certain clock edge. If no clock is specified for either handshake, the transaction is performed asynchronously. If both handshake signals are specified, it is illegal to reset the first active signal before the second signal has been activated. Table 5.8 compares the CoMAP handshake types with common handshaking terms in a scenario where a producer writes data to a consumer.

Role CoMAP Handshake	Producer	Consumer
enableout	valid	acknowledge
stallout	wait	busy
enablein	acknowledge	valid
stallin	busy	wait

Table 5.8.: Comparison with common handshaking terms

The producer indicates available data by asserting *valid*. Alternatively, lacking data is signaled by assertion of *wait*. The consumer sets *acknowledge* while accepting data. As an alternative, the consumer may choose to assert *busy* when it is not ready.

For all handshakes, a time **offset:** (see grammar above) or initial **latency:** with regard to another handshake may be specified. The offset value is given by

$$offset := t_{handshake} - t_{port}, t \in \mathbb{N}$$

with $t_{handshake}$ being the number of the clock tick when the handshake is asserted, t_{port} being the number of the clock tick when the data on the port is valid (outgoing) resp. available (incoming), and t being the clock tick counter of the associated clock port (**clock:**, see Figure 5.8).

The initial latency does not represent a hard timing relation between handshake signals, instead it gives a hint at how much prefetching or buffering should be applied in the data transfer pipelines in both communication peers. It is defined by

$$latency := t_{hs_target} - t_{hs_initiator}, t \in \mathbb{N}$$

with $t_{hs_initiator}$ being the number of the clock tick when the handshake that initiates a transaction is asserted, and t_{hs_target} being the number of the clock tick when the target is expected to respond by asserting its handshake. Note that the initial latency only applies to the first transaction after the initiator has set its handshake from the inactive to the active state. While the notion of a latency is rather obvious when considering communication *targets*, the *initiator* expresses by indicating a latency that it prefetches data and has an input buffer or pipeline of such extent. Hence, an initiator is able to handle a target with a *matching* latency without interruptions in the data stream. Since offset and latency are calculated from clock ticks, both features are only available for synchronous handshakes.

The timing diagrams that are shown in Figures 5.16 and 5.17 illustrate offset and latency. The differences D0 and D2 in Figure 5.16 demonstrate offsets for an **enableout** handshake, while D1 and D3 show a **stallin** handshake type. The relative differences in clock cycles are measured between the **data** beats of the port (shown as A to D) and the corresponding **enableout** and **stallin** handshake signals. Figure 5.17 illustrates the measurement of latencies between the handshake signals **enableout** and **enablein**. The relative differences of clock cycles are valid only for the first response to the assertion of

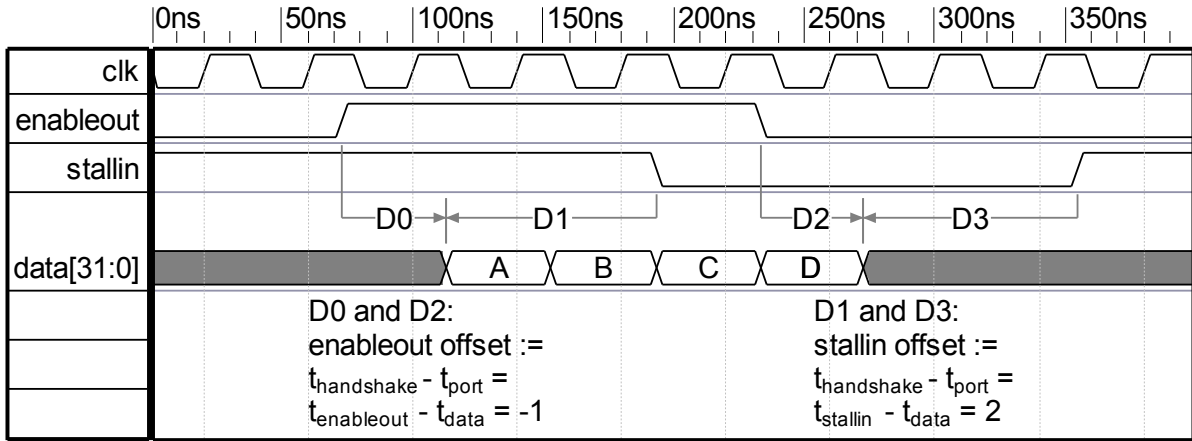


Figure 5.16.: Offsets measured between handshakes and port data

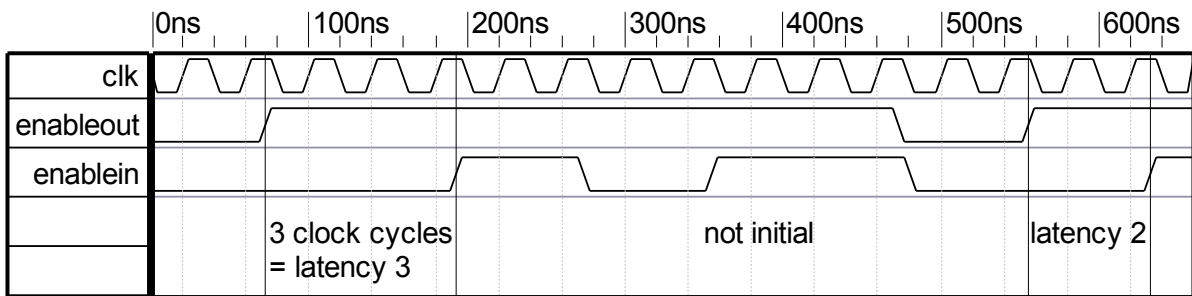


Figure 5.17.: Latencies measured between two handshakes

a handshake signal. Hence, the response labeled as **not initial** in the figure is irrelevant for latency calculations.

In addition to the data semantic, an *interrupt* semantic can be selected for ports by setting the transaction type to **interrupt**. This is useful if no actual data transfer is required during the transaction, e.g., to indicate that data must be fetched from a mailbox register. The meaning of all sequence and handshake parameters remains the same as in a data transaction. The only difference is that no data is actually transferred over an interrupt port. In most cases, interrupt ports are modeled in pairs with handshakes crosslinked to the other partner to form an interrupt-acknowledge scheme. As an alternative, the handshake of an interrupt-induced data transfer can be used to acknowledge this interrupt. During the interface binding process (see Section 5.7), *interrupt controllers* may be inserted on the bus between interrupt ports of interrupting blocks and an interrupt service block.

In the example shown in Figure 5.18, an interrupt applied to input port (`irq`) initiates the fetch of data items over the port `mailbox_read`. The data is transferred synchronously to the assertion of both `irq` and `ack`. Simultaneously to the read operation, the interrupt is acknowledged synchronously to the assertion of `ack` (**offset: 0** in the declaration of port `irq`), whereas the assertion of `ack` always corresponds to the active state of `irq`

```

port irq
  transaction: interrupt
  direction: input
  width: 1
  enableout name: ack offset: 0 latency: 0
end port
port ack
  transaction: interrupt
  direction: output
  width: 1
  enablein name: irq offset: -1 latency: 5
end port
port mailbox_read
  transaction: data
  direction: input
  width: 32
  enablein name: irq offset: 0 latency: 0
  enableout name: ack offset: 0 latency: 0
  ...
end port

```

Figure 5.18.: Handshakes with interrupt and data semantics

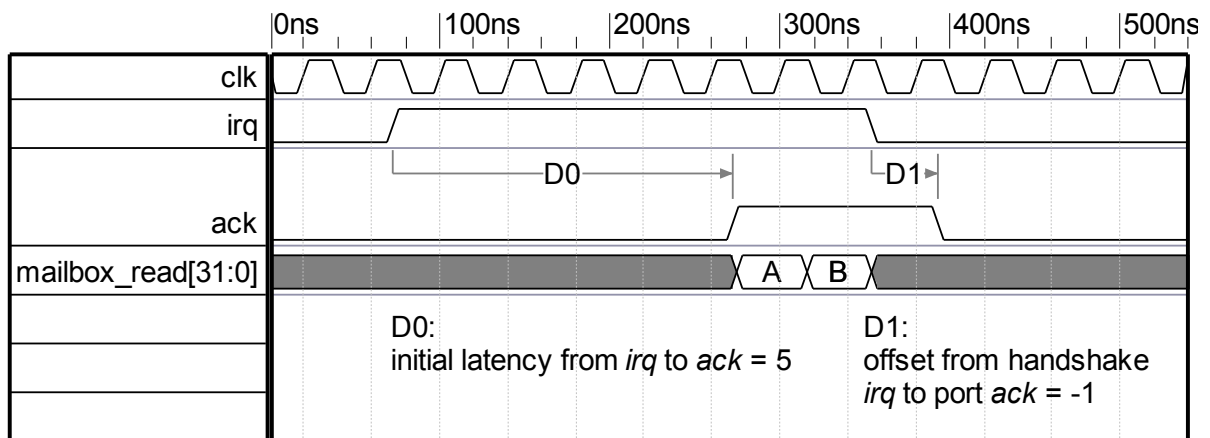


Figure 5.19.: A read with acknowledge triggered by an interrupt

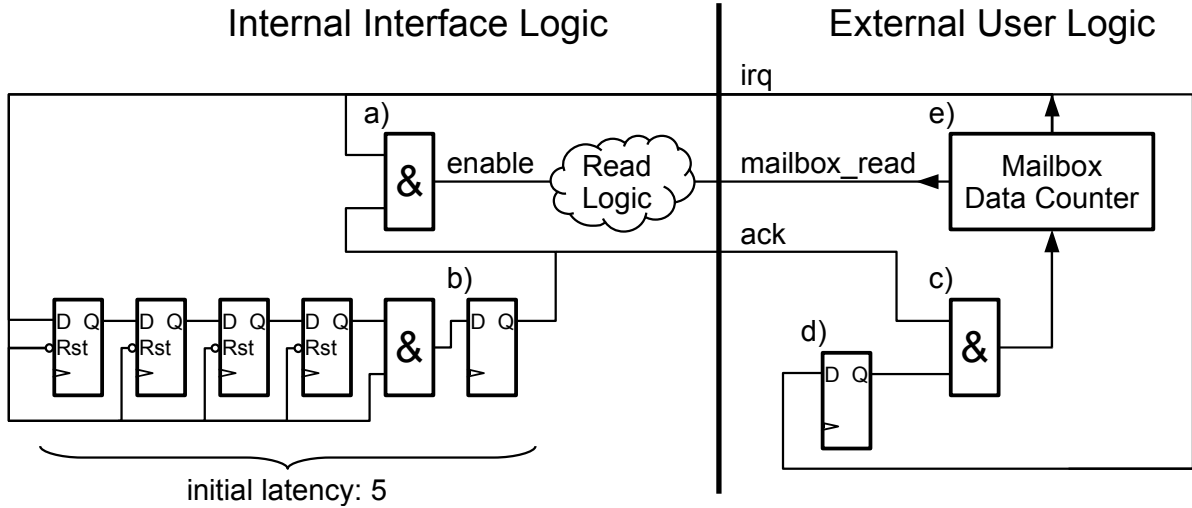


Figure 5.20.: Interface logic implementing the described handshake protocol

one clock cycle earlier (**offset: -1** in the declaration of port **ack**).

The timing diagram in Figure 5.19 presents the same scenario as is shown textually in the example in Figure 5.18. The netlist shown in Figure 5.20 represents interface logic that can handle the described protocol. Since this logic can be generated from the CoMAP description, the combinational circuits for the ports **irq** and **mailbox_read** (via Figure 5.20 a) directly refer to the **offset: 0** (= no delay) statements in the corresponding port declarations. Accordingly, the single flip-flop in the **ack** circuit (Figure 5.20 b) corresponds to **offset: -1**, indicating a single clock cycle delay.

An external user of this interface (i.e., the *producer*, cf. Table 5.8) first has to activate the **irq** signal (see Figure 5.20 e), and then wait for the acknowledgement **ack** to be set. The internal logic on the other side of the interface (i.e., the *consumer*) detects the interrupt request, and responds with the assertion of the **ack** signal, typically with an initial latency of five clock cycles (including the **ack** flip-flop, Figure 5.20 b). With both **irq** and **ack** active (combinational logic, Figure 5.20 a), the internal logic now reads mailbox data via the port **mailbox_read**. Simultaneously (combinational logic, Figure 5.20 c), the external user recognizes the acknowledgement, relates it to the previous clock cycle (Figure 5.20 d), and counts a data item for every read cycle (Figure 5.20 e). It finally resets the **irq** signal when all data has been fetched. In response, the internal logic immediately (combinational logic, Figure 5.20 a) stops the data transfer and resets **ack** one clock cycle later (Figure 5.20 b). Note that the interface logic shown in the figure is just one possible instance with deterministic read behavior. Alternatively, the read logic could choose to suspend reading by temporarily resetting **ack** (not shown here).

The handshaking mechanism discussed here is able to model the dynamic flow of interface protocols while operating without a central flow control authority. When such static interface properties no longer suffice to describe the characteristics of an interface, an enhanced version of the FLAME [Koch07, Koch03] UCODE notation is employed

(described later in Section 6.3.2) to describe dynamic behavior. As UCODEs are also used to coordinate transactions that involve *multiple* interfaces, they are attached to a component (explained later in Section 5.5), which is the superordinate element to interfaces in the CoMAP data model (cf. Section 5.1).

The next section explains the remaining port properties, which focus on bus arbitration and traffic modeling.

5.4.2.2. Arbitration and Data Traffic Characteristics

This section explains the two last properties of a port, **access:** and **traffic:**. The first provides information for automatic generation of bus arbitration logic on multi-master buses, while the second defines the statistical data traffic profile of the port. Such profiles can be exploited to generate optimally adapted memory hierarchies, or for rate matching between interfaces.

Property	Description
access	Port is master, slave, or both
priority	Arbitration priority
request	Port name for bus requests to arbiter
grant	Port name for bus grants from arbiter

Table 5.9.: Access properties

The properties of the **access** element are summarized in Table 5.9. The master capability options of the bus attachment are selected via *access*, while *priority* determines its level of precedence. *Request* and *grant* define the ports which are used to communicate with the arbiter. The grammar for the **access** element is shown below:

```

access ::=
    access: access_type [priority] [request_specifier]

access_type ::=
    master | slave | masterslave

priority ::=
    priority: priority_specifier

priority_specifier ::=
    natural | identifier

request_specifier ::=
    request: port_name grant: port_name

```

The **access:** designator indicates whether the port acts only as bus **master**, **slave**, or both (**masterslave**). The designer may specify in the **request:** - **grant:** option which port is used to request the bus, or receive a bus grant respectively. Note that only

```

port high_1
  access: masterslave
  priority: 5
  request: high_1_req
  grant: high_1_gra
end port
port high_2
  access: masterslave
  priority: 5
  request: high_2_req
  grant: high_2_gra
end port
port low
  access: masterslave
  priority: 0
  request: low_req
  grant: low_gra
end port
port none
  access: masterslave
  request: none_req
  grant: none_gra
end port

```

Figure 5.21.: Bus arbitration between four master-capable ports in two priority classes

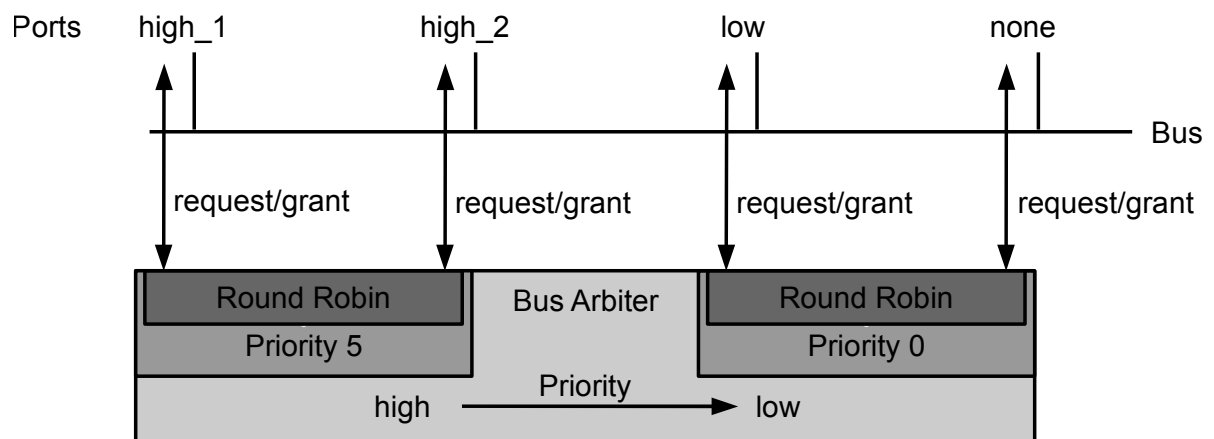


Figure 5.22.: Bus arbiter: Four ports in two priority classes

Property	Description
<code>class</code>	Regular or irregular accesses
<code>distribution</code>	Distribution type in class
<code>gaussian</code>	Percentage under gaussian graph within range
<code>range</code>	Address range for gaussian distribution
<code>local</code>	Local distribution, accesses confined to interval
<code>discrete</code>	Number of sets in discrete distribution
<code>setsize</code>	Set size in bytes
<code>blocksize</code>	linear distribution, data block size
<code>burstiness</code>	Percentage of burst blocks in traffic
<code>burstsize</code>	Burst block size

Table 5.10.: Traffic properties

masters and **masterslaves** may request the bus. An arbiter is generated automatically if more than one master exists on a bus. It grants the bus according to descending priorities, which are selected via the **priority:** option, either as a constant natural value, or dynamically by specifying a port name. This additional port carries a binary-encoded priority level, which provides for flexible runtime arbitration schemes. Ports without **priority:** designator default to the lowest **priority: 0**. If multiple ports on a bus have identical priorities, they are managed by per-priority round robin schemes, respectively. The example shown in Figure 5.21 declares four masterslave ports on a single bus. The port named **none** has no priority declaration. Hence, its priority defaults to zero. All ports are attached via an arbiter, which applies a round robin scheme for identically prioritized ports within the *same* priority class (in this example, **none** and **low** as well as **high_1** and **high_2**, respectively). Among the two priority classes, the arbiter prefers the higher over the lower. Thus, only when no requests are pending from the higher-prioritized class, the arbiter considers requests from the lower. The resulting bus system is illustrated in Figure 5.22.

The **traffic** block defines data traffic characteristics for the port, which can be exploited to optimize the memory hierarchy or the data transfer mechanism that the port interfaces to. This supplementary information does not influence the logical functionality of a component or interface. Notwithstanding, a port should adhere to this information if it is present. Table 5.10 summarizes the properties of the **traffic** block. Irregular or regular access patterns are selected via *class*. The distribution type of the **random** class is determined by *distribution*. *Gaussian* and *range* describe the gaussian distribution type, while *local* defines the local type. Furthermore, *discrete* and *setsize* define the characteristics of a distribution based on discrete sets. Finally, *blocksize*, *burstiness*, and *burstsize* denote the properties of linear-class block-based traffic. The grammar for the **traffic** block is shown below:

```

traffic ::=
    traffic traffic_body
end traffic

```

```

traffic_body ::=
    random_class | linear_class

random_class ::=
    random
    distribution_specifier

distribution_specifier ::=
    distribution distribution_type

distribution_type ::=
    even | gaussian | local | discrete

gaussian ::=
    gaussian: decimal% range: size

local ::=
    local: size

discrete ::=
    discrete: data_set_count setsize: size

data_set_count ::=
    natural

linear_class ::=
    linear
    blocksize: size
    burstiness: decimal% burstsize: size

size ::=
    natural [multiplier]

```

There are two mutually exclusive *traffic classes* which are targeted at caching (**random** class) and streaming (**linear** class) mechanisms respectively. The **random** class provides four statistical **distribution** types: (**even**, **gaussian:**, **local:**, and **discrete:**). The first type, **even**, describes accesses which are evenly distributed over the entire address range, both in locality and time. From such completely random access patterns, no assumptions can be derived about predictable recurring behavior. Hence, special adjustments to the memory hierarchy (e.g., optimized cacheline and cache sizes, or stream engines) may not be effective to increase performance.

The **gaussian:** clause takes two parameters which state that a given fraction (e.g., 80% in Figure 5.23) of the data accesses under a gaussian distribution graph is located

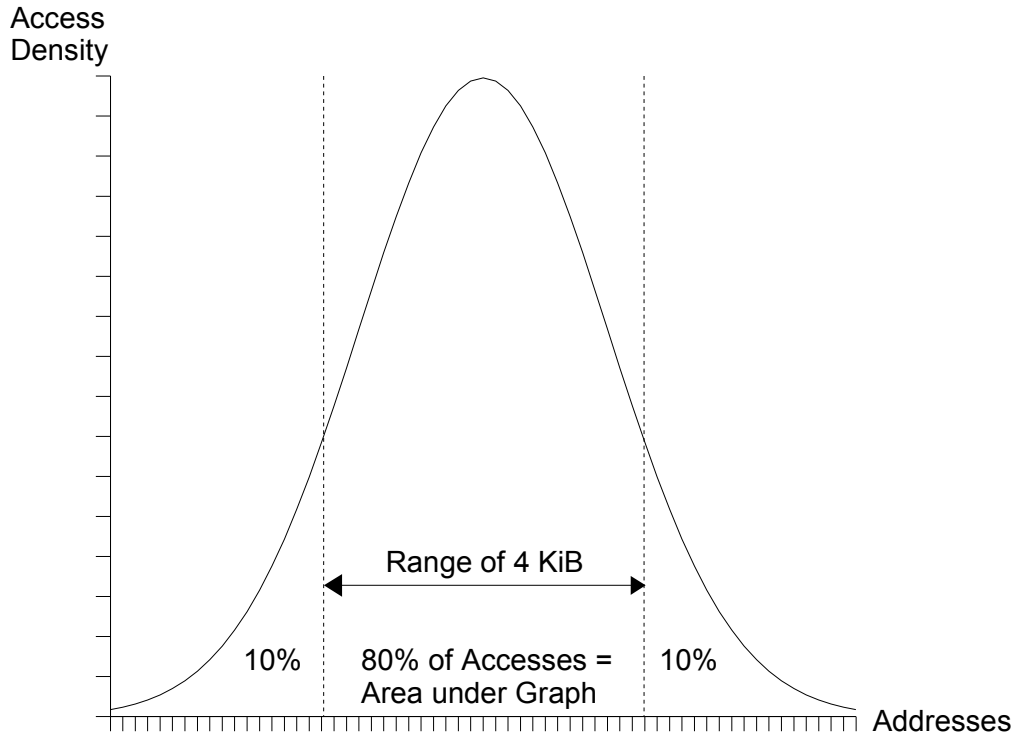


Figure 5.23.: Gaussian traffic distribution

within an address range of the size **range:** (4 KiB in Figure 5.23) in bytes. This address range is centered around the mean value of the distribution. The textual representation of Figure 5.23 is shown in Figure 5.24. The gaussian distribution type denotes the relative locality of data accesses, which can be employed to automatically dimension the cache size for a desired cache hit rate.

```

traffic random
  distribution gaussian: 80% range: 4 Ki
end traffic

```

Figure 5.24.: Example for gaussian traffic scheme (also shown in Figure 5.23)

The **local:** distribution is applied whenever all data accesses are located within an address range of **range:** bytes. Since all accesses are confined to this range, a local distribution can be used for automatic *memory localization*. Localized memories can buffer or store data independently from the main memory, have dedicated ports to transfer data concurrently, and can thus be implemented local to the bus or component that this interface definition refers to. The maximum size of a local memory can be derived from the address range. A local distribution with a range of 4 MiB is shown in Figure 5.25.

The distribution of type **discrete:** states that a certain number of data sets


```

traffic random
  distribution local: 4Mi
end traffic

```

Figure 5.25.: A local traffic scheme

(*data_set_count*) of size **setsize:** in bytes is stored at *data_set_count* fixed address locations (see Figure 5.26). Note that it is not important which data set is stored where, since the data is subject to change anyway. The idea behind the discrete traffic scheme is to provide caching functionality in order to speed up data accesses. In a fully associative cache setup, the cacheline size would be determined by **setsize:**, the number of cachelines by *data_set_count*. With decreasing associativity, more cachelines would be needed (e.g., for a $\frac{\text{data_set_count}}{2}$ -way set-associative cache, the required number of cachelines would roughly double). Figure 5.27 shows an example describing ten discrete sets of data items with a size of 512 bytes each (e.g., corresponding to ten fully associative 512 byte cachelines).

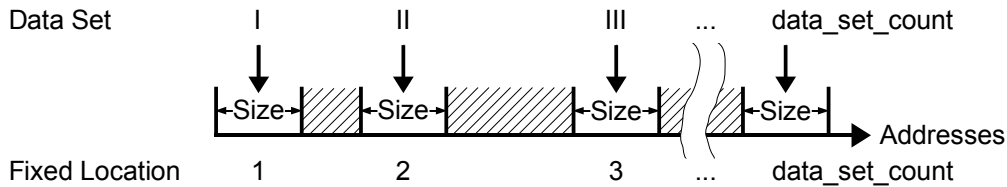


Figure 5.26.: Discrete traffic distribution

```

traffic random
  distribution discrete: 10 setsize: 512
end traffic

```

Figure 5.27.: A discrete traffic scheme

The **linear** traffic class represents a data stream that has a base block or base frame size of **blocksize:** bytes. The **burstiness:** indicates the relative portion of all blocks which exceed the base size. For these blocks, the increased size is defined by **burstsize:**. A single block is expected to be transferred contiguously in consecutive clock cycles, without any pauses by the producer. A data *stream engine* is recommended for handling linear traffic ports at maximum performance. The FIFO sizes for different Quality-of-Service profiles (average, peak load) of this stream engine can be determined from the blocksize and burstiness values. Moreover, those values can be used for rate matching between interfaces. This high-level concept attempts to match or adapt data flow characteristics for different communication peers, and is often found in EDA tools (e.g., Symphony Model Compiler [Syno10b], Xilinx System Generator for DSP [Xili09b])

that are targeted at data processing applications (e.g., DSPs, FIR filters, and multimedia codecs). Apart from CoMAP's traffic characteristics described here, rate matching would apply to the interface binding process (explained in Section 5.4.1), which would have to connect preferably interfaces with similar traffic profiles. However, the details are not elaborated further in this work. Figure 5.28 shows a linear data stream with an equally distributed block size of 128 and 256 KiB (e.g., corresponding to a 384 KiB FIFO for double-buffering).

```
traffic linear
  blocksize: 128Ki
  burstiness: 50% burstsize: 256Ki
end traffic
```

Figure 5.28.: A linear traffic scheme

With interface and parameter definitions now in place, the next section will introduce components, which represent a hardware (or software via a GPP) entity in a CoMAP (r)SoC.

5.5. Components

A component is the basic building block for CoMAP platforms (see next Section 5.6). To this end, it is defined independently from platforms to build a component library for design reuse. It represents a hardware functional unit and usually comprises the definition of several parameters as well as optional interfaces. The latter can be defined directly in a component definition, or may be instantiated as a template.

An overview of the component properties is shown in Table 5.11. A unique key which identifies a component for platform composition is formed by *name* and *version*. *Author/organization* and *comment* provide informal information, while *technology*, *area*, *speed*, and *power* describe implementation details. Furthermore, *behavior* links to the PaCIFIC description for complex protocols that involve multiple interfaces (see next chapter). Parameter interdependencies, which may be required to limit the configuration space of a component to valid parameter combinations, are described by dependency relations (see next section) that are stored in the *dependencies* clause. Preceding the *interface* and *parameter* sections in a component definition (see grammar below), which link to the respective grammar in Sections 5.4 (*interface*) and 5.3 (*parameter*), there is a *component_header* which encapsulates miscellaneous informative as well as technical properties. The design hierarchy of an IP core may be encapsulated in and diversified by an arbitrary number of component definitions, each representing a valid instance of the parameterized master IP core (shown in Figure 5.29). Bus bridges are a special case of IP cores.

Property	Description
name	Name, unique key for platform composition
version	Version number, unique key for platform composition
author/organization	Originator information
comment	Informal comment
technology	Implementation technology or process
area	Implementation area, units depend on technology
speed	Implementation clock speed, depends on technology
staticpower	Static power consumption, depends on technology
dynamicpower	Dynamic power consumption, depends on technology
behavior	Dynamic protocol for multiple interfaces, link to PaCIFIC
dependencies	Dependency relations describe parameter interdependencies
interface	Interfaces of this component, may instantiate templates
parameter	Parameters of this component

Table 5.11.: Component properties

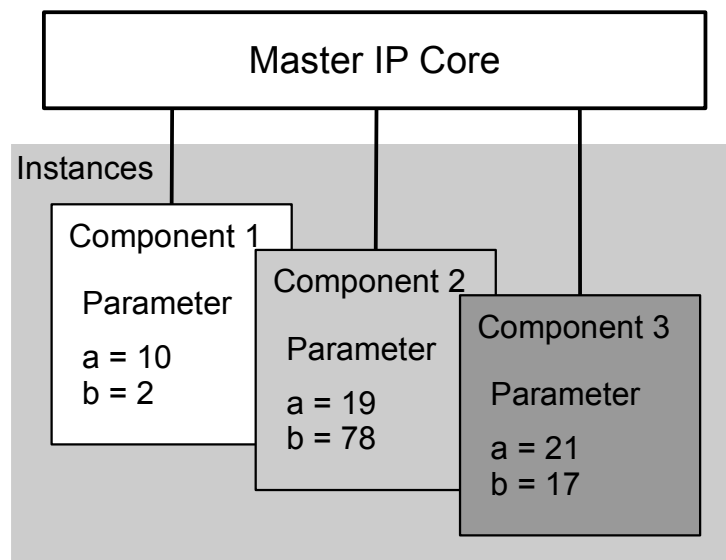


Figure 5.29.: Components derived as instances from a master IP core

5. Platform Management

```
component ::=
    component_header
    [technology_specifier]
    [area_specifier]
    [speed_specifier]
    [static_power_specifier]
    [dynamic_power_specifier]
    [{behavior}]
    [dependencies]
    [{interface}]
    [{parameter}]
    component_end

component_header ::=
    component component_name
    header

component_name ::=
    identifier

technology_specifier ::=
    technology: string

area_specifier ::=
    area
    {string minmax}
    end area

minmax ::=
    min: size max: size

speed_specifier ::=
    speed min: speed_value max: speed_value

speed_value ::=
    pos_float [multiplier]Hz

static_power_specifier ::=
    staticpower min: power_value max: power_value

dynamic_power_specifier ::=
    dynamicpower min: power_value max: power_value

power_value ::=
    pos_float [multiplier]W

component_end ::=
    end component
```

```

component simple
  version: 0.1
  comment
end comment
  technology: "tsmc18"
  area
    gates min: 1500 max: 2000
    srams min: 1 max: 1
  end area
  speed min: 100 MHz max: 150 MHz
  staticpower min: 10 nW max: 12 nW
  dynamicpower min: 0.37 mW max: 1.4 mW
  interface simple
    template: simple_template
  end interface
  parameter data_width
    location: "top_level"
    value: 32
    type: integer
    modifiable: yes
    strippable: yes
  end parameter
end component

```

Figure 5.30.: A simple component definition

The example shown in Figure 5.30 defines a component named *simple*, sets its technology profile, instantiates the interface template *simple_template* for its single interface *simple*, and defines a parameter *data_width*. When instantiated from a platform (see next Section 5.6), a component is identified by its unique key that consists of its name (*component_name*) and its **version:** tag, which is part of the **header** grammar construct. The remaining header parts are described in Section 5.4.

The **technology:** term (see Figure 5.30) indicates the hardware technology or silicon process on which the design that is associated with the component is typically implemented. To accommodate all present technologies and cater for future technology developments, it is defined as *string* with open semantics, which has to be interpreted by the design flow tools or HW designer. The **area**, **speed**, **staticpower**, and **dynamicpower** values relate to this technology. Hence, a matching set of area properties must be specified for each technology. Analogous to **technology:**, these **area** properties are also defined as *string*, and thus have to be interpreted as well. For the remaining parts of this work, it is assumed that technology properties are specified for TSMC's 0.18 micron process [Taiw05] and Virtex-5 FPGAs [Xili10a] (displayed in Table 5.12).

For **speed**, **area**, and **staticpower/dynamicpower**, the value defined by **min:** rep-

Technology	Description	Property	Description
tsmc18	TSMC 0.18 micron process	gates	Cell library logic gate
		srams	Single-transistor SRAM
XC5VFX70T-1	Xilinx Virtex-5 FX	luts	6-input look-up table
		flipflops	CLB flip-flop
		brams	Block RAM
		dsps	DSP48E block

Table 5.12.: Example specification of technologies and their properties

resents a minimum, **max:** is a maximum. The speed maximum and area minimum are achieved by applying mutually exclusive, aggressive synthesis and mapping constraints. These constraints, jointly with the operating conditions, also influence the power dissipation bounds. Component design changes based on parameter modification can result in different area, speed, or power values, which should always reflect all intended configurations. Technology data can be used, e.g., by an SoC builder or the hardware kernel scheduler of a reconfigurable computer.

The *behavior* grammar construct describes FLAME UCODEs [Koch07, Koch03], which are part of the PaCIFIC framework (described in the next Chapter 6). They represent complex interface protocols that cannot be covered by CoMAP's static interface handshaking mechanism (see Section 5.4.2.1 above). A behavior definition links CoMAP to PaCIFIC and enables components to benefit from dynamic C-to-HW interface properties. Since behaviors can coordinate *multiple* interfaces of a component, they are located in the component definition instead of the interface definition.

The next two sections describe how parameter interdependencies can be expressed via dependency relations.

5.5.1. Dependency Relations

Boundary functions (see Section 5.3) define a continuous interval, are local to a parameter, and thus cannot express parameter *interdependencies*. On the other hand, dependency relations represent an alternative means for expressing parameter constraints, including interdependencies. To this end, a dependency relation defines a set of discrete values that are accepted as valid for a *dependent parameter*. More precisely, a *dependency relation* is a set of n -tuples (a discrete relation) which defines valid combinations of $n - 1$ independent parameters and 1 dependent parameter. A *dependent parameter* is a parameter which is required to recalculate its value whenever the following conditions are true:

- The dependent parameter is a member of the dependants list (see Section 5.3, dependants block) of another parameter, and
- the value of the latter has been changed.

```

dependencies ::=
    dependencies
    {dependency_relation}
end dependencies

dependency_relation ::=
    ( parameter_name [{, parameter_name}] ) :
    { n_tuple [{, n_tuple}] }

n_tuple ::=
    ( expression [{, expression}] )

```

The parameter names (*parameter_name*) refer to parameters inside the component definition. The *i*-th parameter in the leftmost tuple (before the colon) corresponds to the *i*-th element of every *n*-tuple on the right side of the colon. Each *i*-th element represents a valid value for the *i*-th parameter. The elements of each *n*-tuple may be constants, parameters, ranges (integer intervals), functions, or any combination of these (complex expressions). A range (integer interval) as constant expression input yields a set of valid parameter values by iterating through the range and evaluating the expression each time. Note that ranges are allowed here in contrast to the boundary functions.

Since dependency relations are a powerful instrument for describing parameter interdependencies, the designer has to take special care in preparing sensible, easily manageable sets of tuples. The examples in the following section clarify how dependency relations work.

5.5.2. Examples for Dependency Relations

The dependency relations which were introduced by the component specification (see previous Section 5.5) are elaborated in this section. Consider a memory system with the following parameters:

- *Size*: The memory size in mebibytes
- *Banks*: Number of parallel memory banks
- *Addr_width*: Width of address bus
- *Data_width*: Width of a data word per bank in bytes
- *Technology*: Type of technology

Assume that technology *X* may be used for memory sizes up to 64 MiB, and technology *Y* for sizes between 32 MiB and 128 MiB.

The dependency relation (in this case a function of *Size*, *Data_width*, and *Banks*) for *Addr_width* is given by

$$(Addr_width): \{(\log(2, Size / (Data_width * Banks)))\}$$

The same function resolved to the number of banks would be

```
(Banks): {(Size / (Data_width * pow(2, Addr_width)))}
```

Other resolutions can be derived analogously.

A constraint for *Data_width* and *Banks* is expressed as

```
(Data_width): {(1), (2), (4)}
(Banks): {(1), (2), (4)}
```

Here, each parameter is defined by a set of singleton tuples which represent valid values that do not depend on other parameters. As an alternative, identical constraints could be expressed using discrete ranges:

```
(Data_width): {(pow(2, 0..2))}
(Banks): {(pow(2, 0..2))}
```

In this example, $\frac{Size}{Data_width \cdot Banks}$ must be a power of 2. Resolved to *Size* that is

```
(Size): {(Data_width * Banks * pow(2, 0..1000))}
```

The values for *Technology* and *Size* interdepend. Hence, if *Technology* equals *X*, *Size* may range from 0 to 64 MiB and vice versa. On the other hand, if *Technology* is equal to *Y*, *Size* must be in the range from 32 to 128 MiB and vice versa.

```
(Technology, Size): {"X", 0..64}, {"Y", 32..128}
```

A dependency relation combines the *n*-th values of all tuples into a common category. In the example above, *Technology*, "X", and "Y" form the first category, whereas *Size*, 0..64, and 32..128 form the second. Each tuple then expresses a single relationship between these categories, e.g., *Technology* "X" and *Size* 0..64 form a valid relationship. The entirety of relationships in a dependency relation enumerates the set of valid parameter combinations.

Thus, dependency relations define valid parameter sets for components, which are the building blocks for platforms. The latter will now be described in the next section.

5.6. Platforms

A platform represents a system environment into which a hardware or software component is integrated. It is composed of one or more platform configurations which are defined as a set of *active components* (see Figure 5.31). An active component is the instance of a parameterized IP or software core that is actually chosen for the system being built. All components as well as their interface templates (described in Section 5.4) are then fetched from the respective libraries in the CoMAP repository. Bus bridges are a special case of components (described later).

The properties of a platform are summarized in Table 5.13. A unique key to identify the platform is formed by *name* and *version*, while *author/organization* and *comment* provide informal supplemental information. Each *configuration* statement represents a

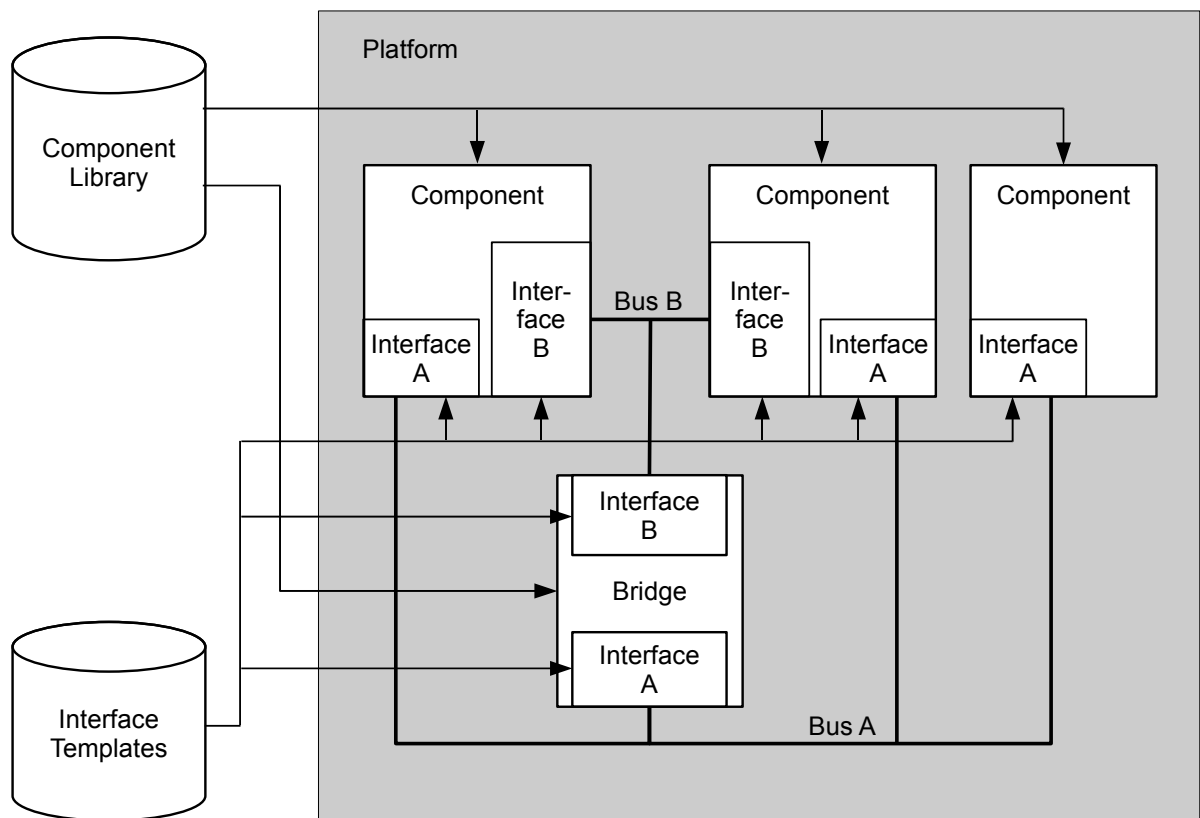


Figure 5.31.: Platform composition using active components and interfaces templates

Property	Description
name	Name, unique key
version	Version number, unique key
author/organization	Originator information
comment	Informal comment
configuration	One set of active components per configuration, multiple configurations
configuration header	Version and informal information
component	Activates instance of component
bridge	Activates instance of bridge component

Table 5.13.: Platform properties

5. Platform Management

valid system composition of multiple components, which are activated by *component* and *bridge* instantiations. Multiple configurations per platform are possible, which can be distinguished by version information stored in the *configuration header*.

After the **platform** keyword and the *platform_name*, the *header* (explained in Section 5.4.1) defines version and author information, followed by the instantiated active components:

```
platform ::=
    platform platform_name
    header
    {platform_configuration}
end platform

platform_name ::=
    identifier

platform_configuration ::=
    configuration_header
    [{component_or_bridge}]
    configuration_end

configuration_header ::=
    configuration configuration_name
    header

configuration_name ::=
    identifier

component_or_bridge ::=
    component | bridge

component ::=
    component: platform_require

bridge :=
    bridge: platform_require

platform_require ::=
    component_name [require: [exact] version_number]

configuration_end ::=
    end configuration
```

The platform configurations are the core part of a platform definition. Several alternative configurations may exist. They consist of a configuration header (which provides a

separate version for each configuration, and informative data), followed by component or bridge instantiation statements, and finally the end delimiter. A component instantiation binds a block, which is associated with the component that is selected by the **component:** statement, with its full functionality to the **platform**. In contrast, the **bridge:** instantiation expects a block which only provides address translation and data transfer between its interfaces. The platform is composed by connecting the interfaces of all instantiated blocks, employing the interface binding mechanism (see Section 5.4.1). Note that some of the active components, which are explicitly instantiated by the platform, may require additional resources (see Sections 5.1 and 5.4.1) that are hence added implicitly to the platform.

```
platform simple
  version: 0.1
  comment
end comment
configuration default
  version: 0.3
  comment
    "basic config"
  end comment
  component: simple_ip require: 2.1
end configuration
end platform
```

Figure 5.32.: A platform definition

In the platform definition shown in Figure 5.32, the optional **require:** *[exact]* extension of the **component:** instantiation statement demands at least (or exactly, if **exact** is specified) version 2.1 of component *simple_ip*. This extension is also available for a **bridge:** instantiation. The platform **version:** in conjunction with its name is a *unique key*, which is used to select an instance of a platform with a certain configuration name and version.

As the CoMAP repository is now completed, several tools which operate on the repository data are presented in the next section.

5.7. CoMAP Tools

A set of tools is specified to work on the CoMAP repository and HW/SW design source files, two of which were realized at an industrial partner. The first category of tools provides support for the system designer in managing the repository contents, setting up and configuring components and platforms as well as extracting and (re)inserting parameters from or into design files. The second category aims to overcome certain

Tool	Description	Design support	Improve EDA flow
CoMAP-VPP	Parameter stripping and parameterized instance generation	X	X
Parameter Extractor	Collect free parameters from design files	X	
Consistency Checker	Verify constraints and dependencies	X	
Address Finder	Find address mappings across bridges	X	
Parameter Editor	Edit parameter properties	X	

Table 5.14.: CoMAP tools and categories

limitations of the Electronic Design Automation (EDA) flow, which will be elaborated below. Table 5.14 shows an overview of the tools and their categories.

EDA tools sometimes generate unpredictable results whenever code inclusion or exclusion statements (`if...generate`, `if (parameter == value) then`) are involved in hardware synthesis, code analysis, or cross translation. In particular, code coverage analysis may deliver unintended results if parts of the code (e.g., optional functions, test logic) are disabled by ordinary program statements which evaluate expressions that are based on parameters. In practice, the proprietary commercial designs which were examined at an industrial partner revealed that such code may be partly or completely flagged by common EDA tools as “not covered”. Moreover, common ASIC synthesis tools needed up to twice the execution time for compiling and synthesizing designs when such critical program constructs were present in the code. Cross translation proved to be incomplete in some cases where parts of the code were parameterized or excluded by the mechanisms mentioned above.

The Hierarchical Preprocessor CoMAP-VPP (CoMAP-Verilog PreProcessor) was implemented on the basis of a proprietary tool at sci-worx GmbH, Germany. The publicly accessible parts of the program are described here in greater detail. CoMAP-VPP builds an instance of a parameterized design by stripping parameters which have been marked as strippable (see Section 5.3), and replacing them with their constant values. This eliminates all incompatibilities due to unused code, varying support for Verilog/VHDL generate statements, and different Verilog/VHDL parameterization features. A commercial requirement is fulfilled as well: Parameter stripping enables an IP vendor to deliver only the specific version of IP core source code that was actually sold. Without parameters, the customer cannot create derivative versions of the IP by simply adapting the parameter set as needed. Furthermore, CoMAP-VPP generates multiple instances of a design module or entity. This is achieved by a preprocessor which unrolls parameterized instantiation statements that are encapsulated in `for` or `while` loops. For every loop iteration, a hard instantiation is produced, providing a replacement for the VHDL generate statement that in some cases incorrectly biased code coverage analysis in a way similar to the effects described above. The missing multiple instance generation support of the older Verilog versions is overcome as well.

To this end, the Verilog PreProcessor (VPP) [ChTJ04] kernel is executed by a Perl

```

/* Input file "top.v" */
// CoMAP-VPP macro, parameter DATA_WIDTH will be stripped
`let DATA_WIDTH 64
module top;
    reg [`DATA_WIDTH-1:0] SOURCE, SINK;
    ...
    data_path DATA_PATH_INST (SOURCE, SINK);
    // CoMAP-VPP macro, parameter WIDTH will be overridden
    // on instance DATA_PATH_INST
    `parameter DATA_PATH_INST:data_path WIDTH=`DATA_WIDTH
endmodule
/* End of input file "top.v" */

/* Input file "data_path.v" */
// Parameter WIDTH will be hierarchically overridden
module data_path(
    input                CLK,
    input    [`WIDTH-1:0] IN,
    output reg [`WIDTH-1:0] OUT);
    // CoMAP-VPP macro
    `if `DATA_WIDTH == 32
        // Optional code will be stripped here
    `endif
    ...
endmodule
/* End of input file "data_path.v" */

```

Figure 5.33.: Sample input files for CoMAP-VPP

class that generates the instance versions of the parameterized design entities and modules. Since the proprietary predecessor tool had been implemented in Perl, the same environment was chosen for CoMAP-VPP. Other Perl-based frameworks for HDL manipulation (e.g., PERLilog [Bill03]) were not mature at the time of implementation. CoMAP-VPP provides Verilog built-in preprocessor style statements including loops, conditional branching, operators, and variables. It also supports hierarchical parameter overriding. Top-level parameter settings are kept in a separate file. Sample input files to CoMAP-VPP (see Figure 5.33) as well as the resulting generated output files (Figure 5.34) demonstrate its parameter stripping, hierarchical overriding, and dead optional code elimination features. CoMAP-VPP has been successfully employed in a commercial IP management tool at sci-worx GmbH (see Figure 5.35).

The Parameter Extractor searches source files for *free parameters*. These are parameters which are not bound to other parameters, neither directly ($x := y$), via a function ($x := 2^y$),

```

/* Output file top-out.v */
// Parameter DATA_WIDTH was stripped
module top;
    reg [64-1:0] SOURCE, SINK;
    ...
    data_path DATA_PATH_INST (SOURCE, SINK);
endmodule
/* End of output file "top-out.v" */

/* Output file "data_path-out.v" */
// Parameter WIDTH was overridden
module data_path(
    input          CLK,
    input          [64-1:0] IN,
    output reg [64-1:0] OUT);
    ...
endmodule
/* End of output file "data_path-out.v" */

```

Figure 5.34.: Output files generated by CoMAP-VPP from input shown in Figure 5.33

nor by hierarchical overriding. The Parameter Extractor handles **generic map** (VHDL), **defparam/#** (Verilog) or similar statements in other languages and automatically resolves the occurring dependencies. All free parameters and, if any, comments are extracted and fed into the CoMAP repository (see Section 5.1) at their correct positions with respect to design hierarchy. Hence, the system designer does not have to start a new design with an empty repository, the automatically generated repository can be merely extended instead. Each source file format is handled by an appropriate back end module. Thus, any new source format can easily be integrated by adding an additional module. At the time of writing, the Parameter Extractor is partly implemented with a VHDL back end. The example VHDL design shown in Figure 5.36 instantiates an entity **example** from the top level, the latter contains a single free parameter **C_DATA_WIDTH**. Figure 5.37 shows the CoMAP repository entry generated from this input by the Parameter Extractor. Note that the parameter comment in the CoMAP output is directly extracted from the VHDL design, and thus is identical.

The Parameter Consistency Checker is specified to verify all constraints and dependencies. It reports parameter settings which do not meet a specific constraint. To this end, it recursively traverses the dependants lists (see Sections 5.5.1 and 5.3) of all parameters and evaluates the boundary functions and dependency relations. It then notifies the designer of any inconsistencies it encounters, providing hints for their resolution.

The Address Finder is defined to recursively traverse bridges on a platform and find address range mappings from an instance at a starting point somewhere on the platform

Reuse library code generator - Konqueror

Location Edit View Go Bookmarks Tools Settings Window Help

Location: <http://localhost:8008>

Please select all parameters

language:

☒ VHDL entity
☐ VHDL architecture
☐ VHDL configuration
☐ Verilog HDL

reset type:

☒ asynchronous reset
☐ synchronous reset

name of module:

input clock:

input reset:

input data:

data width: :0

input valid:

output clock:

output reset:

output data:

output ack:

Figure 5.35.: CoMAP-VPP user interface

```

-- Input file "toplevel_rtl.vhd"
architecture rtl of toplevel is
  -- Free parameter is extracted
  constant C_DATA_WIDTH: integer := 16;
  ...
begin
  ...
  U_MY_INSTANCE: example
    -- Non-free parameters are overridden
    generic map (G_DATA_BITS => C_DATA_WIDTH, G_ADDR_BITS => 8);
    port map (AddrData => AddressData, Outdata => Result);
  ...
end rtl;
-- End of input file "toplevel_rtl.vhd"

-- Input file "example.vhd"
entity example is
  -- Non-free parameters
  generic (G_DATA_BITS, G_ADDR_BITS: positive);
  port (AddrData: in std_logic_vector(G_ADDR_BITS + G_DATA_BITS - 1 downto 0);
        Outdata: out std_logic_vector(G_DATA_BITS - 1 downto 0));
end example;
-- End of input file "example.vhd"

-- Input file "example_rtl.vhd"
architecture rtl of example is
  ...
begin
  ...
  data := AddrData(G_DATA_BITS - 1 downto 0);
  addr := AddrData(G_ADDR_BITS + G_DATA_BITS - 1 downto G_DATA_BITS);
  ...
end rtl;
-- End of input file "example_rtl.vhd"

```

Figure 5.36.: Sample input files for Parameter Extractor


```

comap 1.0
; generated by Parameter Extractor

component toplevel
  version: 1.0
  parameter C_DATA_WIDTH
    location: "toplevel"
    comment
      "Free parameter is extracted"
    end comment
    value: "16"
    type: integer
    modifiable: yes
    strippable: yes
  end parameter
end component

component example
  version: 1.0
end component

end comap

```

Figure 5.37.: CoMAP repository entry for extracted free parameter

to an arbitrary target instance. It is also possible to check whether a given address range mapping from a source to a target can be implemented on a platform with a given bridge configuration. The Address Finder uses the address translation feature of the CoMAP interface definition (see Section 5.4.2 and Figures 5.12–5.13 on pages 84–84) in conjunction with bridge instantiations. For the example shown in Figure 5.38 (cf. Figure 5.39 for the relevant excerpt of the corresponding CoMAP representation), it is assumed that the Address Finder is requested to deliver a valid mapping on the OPB for address range 0x0–0x200 on the PLB. By recursively traversing the bridges, it finds the OPB address range from 0x3E80 to 0x4080. Note that the address space provided by the PCI to PLB bridge (size 0x400) is truncated to 0x300 by the limited internal representation of the OPB to PCI bridge. Hence, a request to map the full PLB range (0x0–0x3FF) would fail.

Platforms and components that are represented in the CoMAP notation are edited either directly with an ASCII editor or by means of a graphical Parameter Editor, which is specified to support formatted entry of actual parameter values, properties, and dependencies into labeled fields. The Graphical User Interface (GUI) also acts as front end, configurator, and project flow manager for the tools described above.

The next section will demonstrate a real rSoC platform represented with CoMAP.

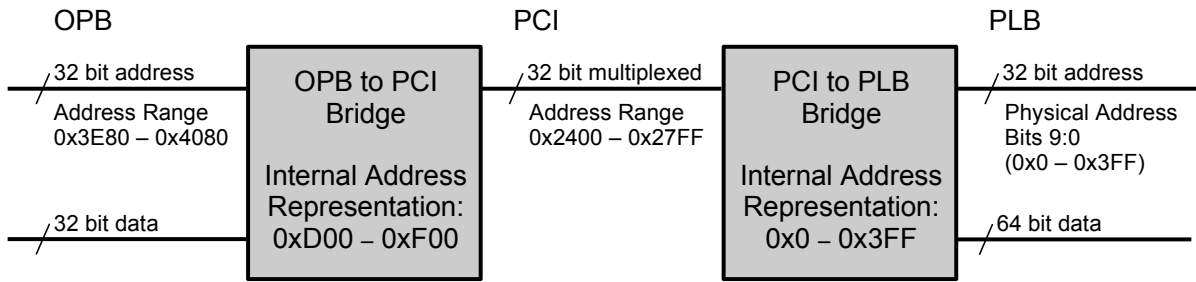


Figure 5.38.: Finding a valid address mapping

```

; OPB to PCI bridge
port OPB_data
...
address port: OPB_addr physical: 0x3E80..0x4080 logical: 0xD00..0xF00
...
end port

port PCI
...
address multiplexed: 0x2400..0x2600 logical: 0xD00..0xF00
...
end port

; PCI to PLB bridge
port PCI
...
address multiplexed: 0x2400..0x27FF logical: 0x0..0x3FF
...
end port

port PLB_data
...
address port: PLB_addr physical: 9:0 logical: 0x0..0x3FF
...
end port

```

Figure 5.39.: CoMAP address mappings for both bus bridges shown in Figure 5.38

5.8. Real World Example: Xilinx ML507 rSoC

CoMAP's capability to describe a real system as well as its advantages will be shown in this section. To this end, the Xilinx ML507 [Xili09c] rSoC example of Section 5.1 is revisited, which represents a simplified, yet fully functional subset (shown in Figure 5.40) of the vendor-supplied ML507 reference design. As before, the platform consists of a DDR2 SDRAM controller attached to external RAM, an interrupt controller, and Ethernet as well as UART interfaces. The PowerPC 440 embedded CPU is connected to memory and peripherals via an internal bus bridge, which provides Memory Controller Interface (Xilinx MCI) [Xili10f, Chapter 5] and Processor Local Bus (CoreConnect PLB) [Inte99] interfaces, respectively. Additionally, the Ethernet interface transfers network frames via two separate DMA links, one for each direction, which are connected to the CPU bus bridge as well. Furthermore, a hardware accelerator (HA) acts as user logic, which is attached via the FastLane+ interface (described in Section 4.5). For the sake of clarity, a simplified PLB representation without advanced features such as bursting is used in this example. Furthermore, the Ethernet MAC (part of the rSoC) and PHY (external) are regarded as a single component, as well as DDR2 controller and the external RAM.

The CoMAP representation of this platform is shown distributed over three listings in the appendix. The HA and all other rSoC elements are defined as parameterized components, which reuse templates (shown in Listing A.1 on pages 223–235) for their interfaces. Likewise, the component definitions themselves constitute a component library (Listing A.2, pages 235–256), which can be reused by other platforms. The actual ML507 platform is defined in Listing A.3 (page 256). Note that the technology definitions shown in Table 5.12 are reused here.

The interface templates comprise the PLB master and slave interfaces including a dedicated clock and reset part, however, the interrupt scheme is declared separately and managed by the interrupt controller. The Xilinx DMA LocalLink [Xili05c] interface templates are provided for both source and destination sides using a sequence definition

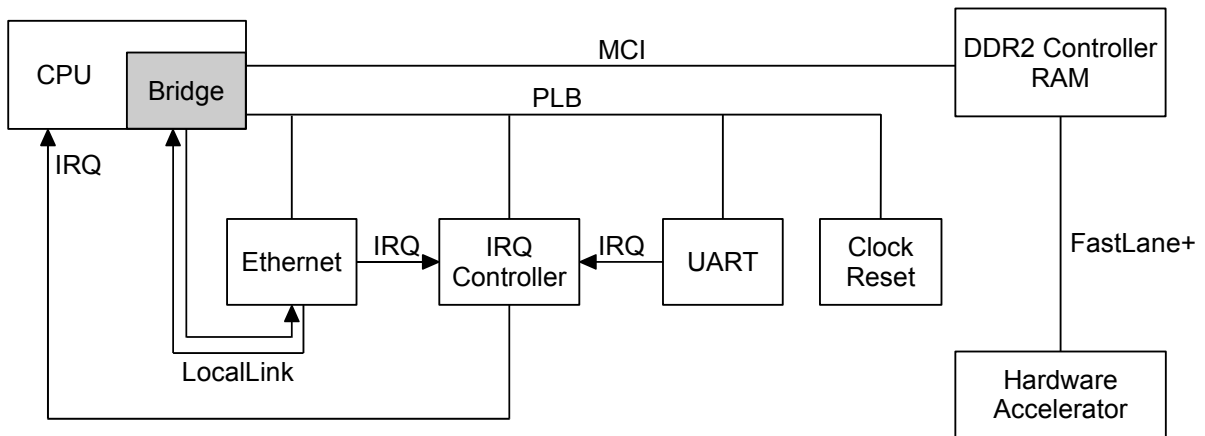


Figure 5.40.: Subset of Xilinx ML507 rSoC reference design

for the LocalLink pair that is used for sending Ethernet frames.

For receiving frames, the template instantiations in both CPU and Ethernet components overwrite the sequence definition with one that matches the receive frame protocol. Since they are never reused, the FastLane+, MCI, and interrupt interfaces are not defined as templates, but directly in the respective component declarations. The clock and reset generator connects to the PLB for signal distribution. Interrupt requests from the Ethernet and UART peripherals are managed by the interrupt controller, which then passes the request to the CPU.

Note that the DDR2 controller is not explicitly part of the platform definition as it is declared as resource. However, it is implicitly added during the interface binding process (described in Section 5.4.1) since it is requested by both CPU and HA.

5.9. Chapter Summary

This chapter presented a platform and configuration management system capable of representing and efficiently handling the high-performance Reconfigurable Computing Platform that this thesis aims to realize. Although not part of the run-time logic, support for automated platform management proved indispensable for even medium-scale platforms, which already comprise many components, buses, interfaces and large parameter sets. Moreover, a well-defined target platform was found to be a prerequisite for automatic hardware/software compilation. To this end, the CoMAP repository was introduced to hold the definitions for interfaces, components and their parameters, and platforms. Mechanisms to automatically connect interfaces and buses to compose a working platform were developed. The requirement to express and validate complex parameter sets and interdependencies was addressed and identified as a key to automatic platform management. Hence, tool support was introduced to assist the system designer with extracting, processing, and inserting repository elements as well as relieving parameter-related limitations of common electronic design automation tools. Finally, a real world example was shown which demonstrates the effectiveness of both platform description and composition, and the interoperability with the other parts of the Reconfigurable Computing Platform described in this work.

6. Interfacing C Language Software and IP Core Hardware

What are the characteristics of the HW/SW interface? How can such an interface be made available to automatic HW/SW compilation and platform composition without restricting its functionality?

Complex (r)SoC and platform-based designs require integration of configurable IP cores from multiple sources. Even automatic compilation flows from a high-level description to HW/SW systems can benefit from having access to reusable sophisticated hand-optimized IP cores. This is especially the case in the domain of reconfigurable computers, which offer core integration directly into the custom datapath [Koch09]. With the powerful execution model developed in this work (see Chapter 4), which provides for tightly-coupled HW/SW interaction, as well as a repository (see previous Chapter 5) that is capable of efficiently representing complex IP cores, it would be beneficial to use and combine both solutions from a high-level software programming language.

This chapter describes the Parametric C Interface For IP Cores (PaCIFIC) [LaKo04], which allows the automatic embedding of complex IP cores in a high-level language such as C. The structure of the PaCIFIC framework will be illustrated by an example which shows the interplay of the mechanisms. After that, the integration of PaCIFIC into the HW/SW compiler Comrade, its execution model (see Section 4.1), and design flow is explained.

PaCIFIC provides for formal description of IP behavior and interface characteristics as well as an idiomatic programming style, thus establishing an automatic design flow that presents convenient, simple C interfaces (function prototypes) to a software programmer inexperienced in hardware design. It hides the formal descriptions of IP- or platform behaviors as well as interface characteristics by encapsulating them with other IP configuration data in the CoMAP repository (described in the previous Chapter 5).

This approach applies not only to a single compiler, design flow, or the specific domain of adaptive computing systems, but generally to all HW/SW co-design environments. The unified notation for IP configuration and interface protocol description enables (semi-) automatic design composition. Reusable interface descriptions allow the separation of interfaces and implementation details. Hence, PaCIFIC/CoMAP is applicable to the entire spectrum of (r)SoC, platform-based and derivative designs.

6.1. Problem Description

Consider a scenario with two IP cores which should be arranged forming a pipeline. Assume that each core has one input and one output interface.

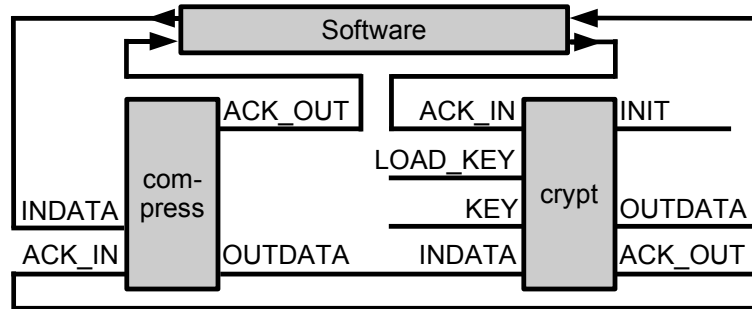


Figure 6.1.: Hardware pipeline used by software

As shown in Figure 6.1, the datapath comprising both cores is supposed to be used from a software description that also sources and sinks the data. A natural approach for plain software would consider the two IP cores to be C functions, leading to the code in Figure 6.2.

```
int *indata, *outdata, *intermediate;
for (n = 0; n < 64; n++) {
    compress (indata++, intermediate);
    crypt (intermediate++, outdata++);
}
```

Figure 6.2.: IP cores as C functions

From such a code description, the HW pipeline shown in Figure 6.1 should be automatically inferred. This requires additional information about the hardware “functions” `compress` and `crypt`. The software developer should not have to be aware of the actual mechanisms that are involved in realizing the structure.

To this end, several issues must be addressed when dealing with hardware that is embedded in a software description:

Recognition of IP cores

Since C cannot distinguish between hardware and software, it has to be detected which of the function calls aim at IP core instances.

Multiple hardware instances

In order to exploit the benefits of parallel hardware execution, it is necessary to distinguish different instances of the same IP core. C does not provide any notion of instances or parallel execution.

Low-level interface control

In contrast to HDLs, plain C has no notion of timing- or cycle-accurate execution schedules. Thus, for each IP core, interface parameters like signal timing, handshaking, and bus arbitration must be provided in an external representation.

Data transfer

There are several ways to exchange data between software and hardware. IP cores are often programmed via register files. Thus, a Programmed I/O (PIO) mode is mandatory in this case. On the other hand, this is highly inefficient for the large data sets which are commonly processed by complex IP cores (video, networking). In these cases, Streaming I/O (SIO) mechanisms are generally employed, often assisted by rate matching and buffering using FIFOs. Such a setup is commonly referred to as a *stream engine*. For each use of an IP core, the appropriate data transfer method has to be determined based upon data-traffic characteristics and interface descriptions that are delivered by the IP provider.

Hardware events

Some transactions are initiated not by the software, but by the IP core, e.g., when the latter is ready to process the next data block. Asynchronous events such as interrupts or error notifications are beyond the semantics of a C function. The functional synchronization, such as the indication of the current state of a hardware function, must be realized, for example, to determine the end of a C function call (=IP core execution) and proceed with the rest of the program.

Constraints and operating conditions

To guarantee correct operation of an IP core, some design constraints (e.g. configuration data, clock speed, I/O timing, maximum length of internal signal paths, technology-dependent parameters, etc.) must be obeyed, which are supplied by the IP provider as part of the IP core description. The constraints are consulted when synthesizing the HW/SW interfaces.

The next section will introduce PaCIFIC's system model, which reflects the considerations mentioned above.

6.2. System Model

PaCIFIC is built on the Configuration Manager for Abstract Parameterizations (CoMAP, see Section 5 above), which is a data model and human-readable description language for the characteristics of individual IP cores as well as entire platforms (shown on the right in Figure 6.3). This section, however, only concentrates on the scope of IP cores.

PaCIFIC consists of rules for an idiomatic programming style which must be used when embedding IP cores in a C source program, and interface control semantics which describe the interface behavior of an IP core (see Figure 6.3). All of the components are tied together in a number of dedicated compiler passes that perform the necessary analysis and synthesis steps (both hard- and software, see Section 6.5).

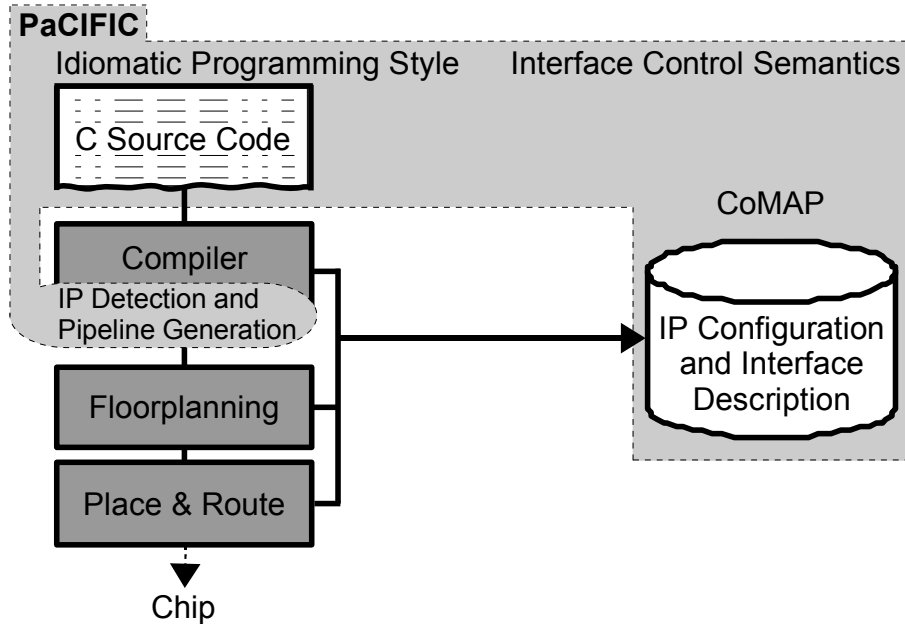


Figure 6.3.: PaCIFIC system model and design flow

The data models and representations are based on the study of more than thirty commercial IP cores (an excerpt which for clarity comprises only the main interface and resource categories is shown in Table 6.1), that were classified using the attributes of the CoMAP interface template (see Section 5.4). The aim was the capability to describe all of the IP cores' interface semantics with the existing attribute catalog. The majority of the evaluated cores belong to the domains of multimedia and networking. The first cores generally presented a datapath oriented interface, with the video or audio stream processing being the main task. In contrast, the networking IP cores employed a processor-based register interface. More complex IP cores even use multiple different interfaces of both kinds.

IP core	Interface	Category	Resource
MPEG2 AC3 Audio Decoder	general	Other	exclusive
	audio data	Streaming	
	host control	Register	
	4 banks ext. RAM	Memory	
Inter IC Sound Receiver	general	Other	
	audio data in	Streaming	
	audio dist module	Streaming	
Inter IC Sound Transmitter	general	Other	
	audio data	Streaming	
	host control	Register	

Table 6.1.: IP core interface classification (continued on next page)

IP core	Interface	Category	Resource
Sound Transmitter 6 Ch.			
as above, additional ports:	audio data ext	Streaming	
Reed Solomon Decoder	general	Streaming	
Viterbi Decoder	general	Other	
	input	Streaming	
	traceback	Streaming	
	core RAM	Memory	exclusive
FEC for DVB-S / DSS	general	Other	
	QPSK	Streaming	
	mode conf	Register	
	code rate conf	Register	
	rate search status	Streaming	
	viterbi RAM	Memory	exclusive
	frame sync conf	Register	
	frame sync stat	Register	
	deinter RAM	Memory	exclusive
	reed solomon conf	Register	
	reed solomon stat	Streaming	
	reed solomon RAM	Memory	exclusive
	descramble conf	Register	
	output conf	Register	
	MPEG2 demux	Streaming	
QAM Demodulator	general	Other	
	restart	Register	
	analog AGC	Streaming	
	output	Streaming	
	test	Register	
	config	Register	
	status	Register	
QPSK Demodulator	general	Other	
	frontend	Streaming	
	FEC	Streaming	
	restart	Register	
	input format	Register	
	AGC conf	Register	
	DC offset comp	Register	
	search mode	Register	
	rate search	Register	
	carrier search	Register	
	lock det conf	Register	
	rate loop conf	Register	

Table 6.1.: IP core interface classification (continued on next page)

6. Interfacing C Language Software and IP Core Hardware

IP core	Interface	Category	Resource
Turbo encoder	carrier loop conf	Register	exclusive
	carrier noise	Register	
	search status	Streaming	
	rate result	Register	
	carrier result	Register	
	variance result	Register	
	DC offset result	Register	
	carrier noise result	Streaming	
	general	Other	
	external	Streaming	
Turbo decoder	RAM	Memory	exclusive
	general	Other	
	external	Streaming	
PCI Initiator/Target	RAM	Memory	exclusive
	pci	Other	
	EPROM	Streaming	
	initiator rd FIFO	Streaming	
	initiator wr FIFO	Streaming	
	DMA	Register	
	target wr FIFO	Streaming	
	target rd FIFO	Streaming	
	sideband	Register	
	general	Other	
CAN Protocol core	CPU	Register	exclusive
	RAM	Memory	
	CAN	Streaming	
	general	Other	
AES En-/Decryption	key/mode	Register	
	plaintext	Streaming	
	cyphertext	Streaming	
	general	Other	
USB	transceiver	Streaming	exclusive
	CPU	Register	
	DPRAM	Memory	
	DMA	Streaming	
	additional I/O	Register	
10G Ethernet MAC	general	Other	
	host	Register	
	tx frame	Streaming	
	rx frame	Streaming	
	XGMII	Streaming	

Table 6.1.: IP core interface classification (continued on next page)

IP core	Interface	Category	Resource
10/100M Ethernet MAC same as above	MII management	Streaming	
	tx rx stats	Streaming	
ATM UTOPIA	general	Other	
	UTOPIA	Streaming	
	ATM	Streaming	
	config	Register	
AAL2 CPS Framer	RAM	Memory	exclusive
	general	Other	
	SSCS	Streaming	
	UTOPIA	Streaming	
	VCI Periheral	Register	
	memory	Memory	exclusive
	config	Register	
	AAL5	Streaming	

Table 6.1.: IP core interface classification

For a detailed description on how these kinds of interfaces can be efficiently realized in a real system, regarding both hardware and software aspects, see Chapter 4. The next section maps the hardware interface characteristics that were found during the IP core study to an interface description which is suitable for both simple and complex protocols.

6.3. Hardware Interface

The CoMAP interface description (see Section 5.4) is used to define the static properties for all IP interfaces as well as the dynamic flow of the interface protocols based on self-arbitrating logic. The latter term describes synchronous logic which operates without a central flow control authority. As usual, properties are expressed as attributes and values. Some of the many defined attributes are (see previous Chapter 5 for complete information):

- Identification (class, type, version, name)
- Auxiliary information (author, comments)
- Port definitions (transaction type, direction, width, associated clock, abstract data type, associated address, handshaking protocol, data traffic characteristics, bus arbitration)
- External resources required by the IP core and their allocation modes (shared, exclusive, persistent). This might include external memories or special I/O requirements (e.g., access to multi-Gbps transceivers)

Port transaction types and handshaking protocols will be examined in more detail in Section 6.3.1. A fragment of a CoMAP interface template for the crypt IP core is shown in Figure 6.4.

```
interface crypt
  type: custom
  version: 1
  port INDATA
    transaction: data
    direction: input
    width: 32
    sequence repeat: inf
      bigendian: 32 bit signed
    end sequence
    enableout name: ACK_OUT offset: 0 latency: 0
  end port
  port ...
  ...
end interface
```

Figure 6.4.: Part of CoMAP interface for the crypt IP

The abstract data type, which is defined by the **sequence** block of the example in Figure 6.4 (here just a single scalar integer), plays a central role in the data exchange between software and hardware. It arranges the nature, order, and count of the data items that are transferred over the port or bus. Every sequence block corresponds to formal parameter of the fictitious C function that represents the IP core.

Interface *templates* can be used to group and reuse the same or similar interfaces in a fashion analogous to the classes and inheritance of object-oriented programming.

6.3.1. Static Interface Properties

The fundamental mechanism for describing self-arbitrating synchronous logic is a handshaking scheme which consists of an incoming and an outgoing signal per port. Since static interface properties have already been defined in CoMAP, this section recalls the essential parts for PaCIFIC of the detailed elaboration in Section 5.4.2:

For an outgoing signal, the asserted state (selectable as high or low) means that the IP core is ready to consume data (on an input port) or that data is waiting to be fetched (on an output port). The incoming signal is the outgoing signal from the connected port at the peer end of the communication. It is not necessary to specify both signals, a one-way handshake is possible as well as no handshake. A transaction is considered complete when all specified handshake signals are active at a clock edge. If both signals are specified,

it is illegal to reset the first active signal before the second signal has been activated. For all handshakes, a time offset or initial latency with regard to another handshake may be specified. Additionally, an interrupt semantic can be selected for the handshaking signals. This is useful if no actual data transfer is required during the transaction, e.g., to indicate that data must be fetched from a mailbox register.

6.3.2. Dynamic Interface Properties

When such static interface properties no longer suffice to describe the characteristics of an IP core's interface, an enhanced version of the FLAME UCODE notation is employed [Koch07] to describe dynamic behavior. UCODEs describe complex interface protocols that exceed the capabilities of the static CoMAP handshake mechanism. The UCODE syntax has been changed for better handling by humans. The unwieldy nested object trees used by FLAME for inter-tool communication have been replaced by blocks of statements (see UCODE grammar below), as known from CoMAP.

```

behavior ::=
    behavior behavior_name
    {ucode}
    end behavior

behavior_name ::=
    identifier

ucode ::=
    proc
    ucode_body

proc ::=
    proc proc_name ( proc_return [{, proc_formal}] )

proc_name ::=
    identifier

proc_return ::=
    identifier

proc_formal ::=
    identifier

ucode_body ::=
    [{ucode_statement}]
    [ucode_exception]
```

```

ucode_exception ::=
    exception
    [{ucode_statement}]

ucode_statement ::=
    ucode_level /
    ucode_posedge /
    ucode_negedge /
    ucode_continue /
    ucode_transfer /
    ucode_start /
    ucode_restart

ucode_level ::=
    level {ucode_portval}

ucode_portval ::=
    ucode_lportval = ucode_rportval

ucode_lportval ::=
    proc_formal / port_name

ucode_rportval ::=
    proc_formal / port_name / parameter_name / integer

ucode_posedge ::=
    posedge {ucode_portval}

ucode_negedge ::=
    negedge {ucode_portval}

ucode_continue ::=
    continue [continue_timeout] [{continue_error}] {ucode_portval}

continue_timeout ::=
    timeout: natural

continue_error ::=
    error: ucode_portval

ucode_transfer ::=
    transfer integer identifier
    [{ucode_statement}]

```

```

    end transfer

ucode_start ::=
    start

ucode_restart ::=
    restart

```

UCODEs are organized as a hierarchy of code blocks. A UCODE block is a list of statements most of which are executed sequentially (except for the handling of pipeline control flows). It represents the state machine of an interface controller. The top-level block in the hierarchy, called **behavior**, is linked to a CoMAP component declaration (see Section 5.5). Multiple **behavior** blocks per component declaration are treated as *parallel* structures, which represent component functionalities that can be executed *simultaneously*. A **behavior** comprises one or more algorithmic interface protocol descriptions, called **proc** statements, which in contrast represent *alternative* operating modes for the execution thread that is represented by this **behavior**.

Each UCODE statement represents a logical transition in the interface protocol and can be directly mapped to hardware. A summary of the UCODE statements is shown below:

The **level** statement asynchronously sets ports to the values given as arguments of the form **port=value**.

The **posedge** statement is similar to **level**, but sets ports synchronously with, and in time (t_{setup}) before a rising clock edge.

The **negedge** statement operates synchronously with a falling clock edge, respectively.

The **continue** statement takes three kinds of parameters: an optional **timeout:n**, optional **error:port=value** expressions and normal **port=value** expressions which are interpreted as conditions. The first two branch to the **exception** block either if all error conditions are true or the timeout in clock cycles has expired. If no timeout or error occurs, the control flow is halted until all normal continue conditions are valid. As stated in [Koch07], multiple conditions in the same **continue** statement are logically ANDed, multiple successive **continue** statements are ORed. The asynchronous **continue** statement can be synchronized by a following **posedge**.

The **exception** block, if present, is located at the end of the UCODE block. It marks the branch target for all **error** and **timeout** clauses and puts the interface or IP block into a well defined error state. The normal control flow terminates if the **exception** block or the end of the UCODE block is reached.

The mandatory **transfer n name** block represents the transfer of **n** sequences to port **name**, with the nature of the sequence being defined in the associated CoMAP interface description (see Section 5.4.2). Without a sequence description on a port,

transfer indicates *n* scalar transfers using the full port width. It acts as a loop in the UCODE control flow. Each iteration is triggered by the handshaking protocol that has been defined for the port. Note that **transfer** is an extension to the standard UCODE repeat operator “*” [Koch07] and additionally links it to a port or sequence. The latter is enabled by allowing multiple statements in the loop body.

The **start** [*n*] statement marks the beginning of a code section where all transfers are pipelined. The optional parameter *n* extends the standard UCODE **start** [Koch07] by a repeat count, which defines the total number of items that are inserted sequentially into the pipeline whenever the control flow executes a **start/restart** section.

The **restart** statement marks the end of the pipelined section. The control flow is forked, creating two threads. The first thread continues sequentially as usual beyond the **restart**, whereas the second thread branches back to the **start** statement, creating a loop (see Figure 6.5). Every iteration (their number is optionally defined by the parameter *n* of **start** above) of this loop inserts a new data item into the pipeline, while another item is drained by the non-branching sequential thread. The section between **start** and **restart** models the steady state of the pipeline, framed by its prologue and epilogue. The latter two require explicit handling outside the **start/restart** construct. As all other UCODE, the **start/restart** repeat count can be directly synthesized to HW analogously to the repeat operator “*” by driving the **LastIn** signal of the pipeline (shown in Figure 6.6) from a counter attached to a comparator. The UCODE pipelining model, which is not specific to the PaCIFIC interface mechanism and beyond the scope of this work, is described in greater detail in [Koch07, Koch03].

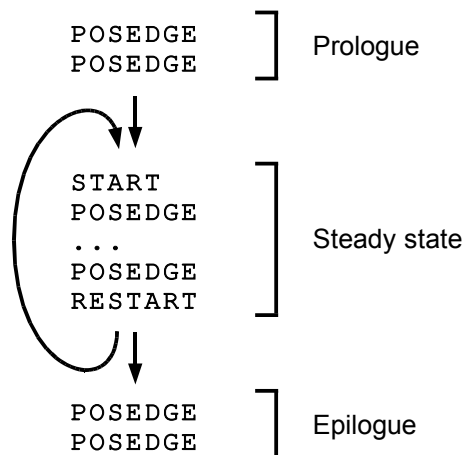


Figure 6.5.: UCODE pipelining model (from [Koch07])

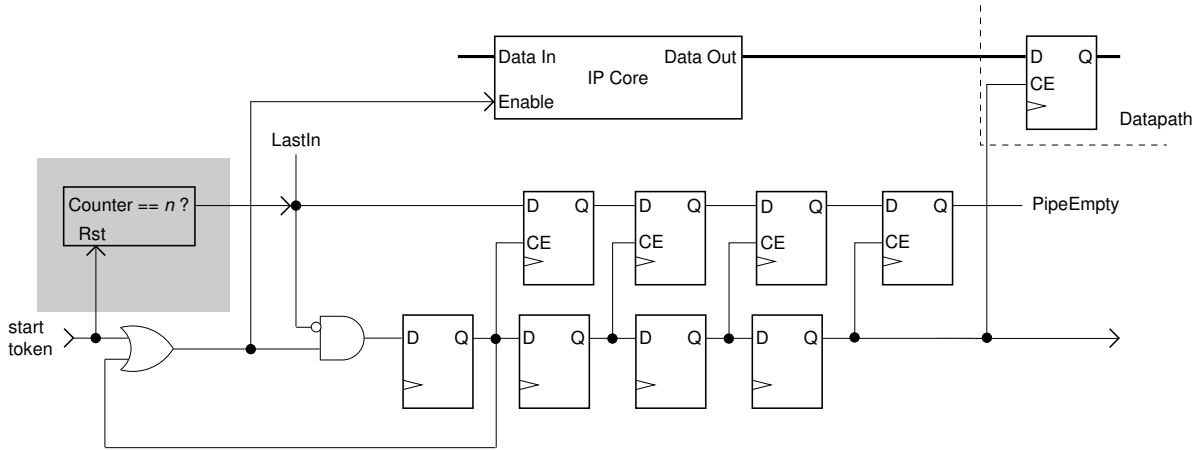


Figure 6.6.: UCODE pipeline administration logic for an IP core with pipeline depth 4, PaCIFIC extension is a repeat counter (based on [Koch07])

Note that **proc** statements in different **behavior** blocks may collaborate in modeling complex protocols. In this case, the designer has to provide for explicit synchronization of these *parallel threads*. Furthermore, explicit handshaking has to be programmed into the UCODE algorithm to replace the CoMAP port handshaking schemes (Section 5.4.2), which are considered merely informative if a UCODE description exists for a port. Nevertheless, both descriptions should not contradict each other.

All UCODE statements are described in greater detail in the FLAME User's Guide and Manual [Koch03]. The following section presents a software interface which complements the hardware interface schemes described above.

6.4. Software Interface

The last sections dealt with the hardware realization of the interface to the IP cores. In this section, the corresponding software mechanisms will be examined.

From the study of multimedia IP cores, it is obvious that a powerful data streaming service is needed to source and sink the datapath interfaces of the IP. The stream engine fetches and stores data from and to shared memory respectively, which is accessible to the SW running on the GPP. The start address of the memory range to be streamed can be expressed as a pointer to C structures which reflect the composition of the sequences defined using CoMAP. Bulk streaming data as well as random accesses to data in that memory range, which are required by some IP cores operating in master-mode, are served via MARC (see Section 4.5.1) and FastLane+ (Section 4.5).

In all cases, the IP cores also require programming (e.g., for initialization) using a register interface. This can be realized by simply mapping the registers into a SW accessible memory region (but not necessarily the main memory space). PaCIFIC regards IP core register contents as live variables, which are transferred via the fast live variable exchange described in Section 4.4.

To recognize the actual IP core embedding and establish both communication methods, an idiomatic C programming style is required. Only two modes of instantiating IP cores from C are supported by PaCIFIC, but they are sufficient to cover all interface types under discussion:

First, there is fully automatic interface generation, which results in the creation of read and write *primitives* for access to the ports of the IP core in both direct (register) and streaming fashions. This method works from the CoMAP interface definition, the IP designer (or more precisely, the author of the CoMAP description) does not have to provide any additional data. However, the SW has to explicitly call the primitives in the required order to actually get the IP core to perform the desired function.

Second, there are functions which atomically perform complex operations without requiring incremental prodding by a SW program. For the realization of these *monoliths*, the designer has to supply an algorithmic description of the control and data patterns that must be applied to the interfaces of an IP core for the required function. The monoliths are then generated automatically. Their call resembles conventional C library functions (all individual control steps have been hidden and implemented automatically).

In both communication methods, the switching between HW/SW execution adheres to the mechanisms of the execution model described in Chapter 4. Note that threading models such as the POSIX one are compatible with PaCIFIC, enabling the parallel execution of HW and SW. This can be beneficial when calling data-intensive IP cores: E.g., while the HW is still running, the SW prefetches the next data block into memory and writes processed data to disk.

6.4.1. Primitives

Consider an input port `indata` without an associated address that is 32 bits wide (cf. crypt IP core interface, Figure 6.4 in Section 6.3). For this case, the C function `write_indata` is generated. It writes 32-bit integers (**sequence bigendian: 32 bit signed**) with a data-dependent termination criterion (**repeat: inf**, models a variable-length data sequence):

```
void write_indata (int *data);
```

A best match approach is employed for mapping scalar hardware data to C data types. The n least significant bits of the next larger C type represent a hardware scalar of width n .

Unrelated to the previous example, an output port with an associated address that delivers a sequence of composite data items (here mapped to the **struct comp**) results in the following function:

```
void read (int *address, struct comp *data);
```

If the **repeat** value in a sequence definition equals one, SW wrapper functions may be used to eliminate the unwieldy pointer in favor of just passing scalar data (e.g., convert ***address** to a plain **int** in the function above). Primitives are most suitable for use with simple register- or memory-style interfaces. More complex interfaces can be realized with monoliths, which will be discussed in the next section.

6.4.2. Monoliths

Due to the strictly sequential semantics of C, it is not possible to directly describe pipelined accesses using primitives. However, this is achievable using monoliths.

The example in Figure 6.7 reconsiders the compress-crypt scenario from Section 6.1 and describes the underlying control protocol for the behavior “encrypt” in PaCIFIC-extended UCODE [Koch07]. The function prototype in the **proc** statement corresponds to the C function, with variables being passed by reference following the rules described in

```

; the crypt function
behavior encrypt
proc crypt(plaintext, ciphertext)

; load key
posedge LOAD_KEY=1
      KEY=0x10D02A19B78D2117 ; fixed key
posedge LOAD_KEY=0

; process single data item
transfer 1 INDATA
  level    INDATA=plaintext
           ACK_IN=1
  continue timeout: 16
           error: INIT=0
           ACK_OUT=1
  posedge ciphertext=OUTDATA
  level    ACK_IN=0
end transfer

; wait for end of pipeline flush
exception
continue INIT=1

; execution terminates here
end behavior

```

Figure 6.7.: UCODE for behavior “encrypt”

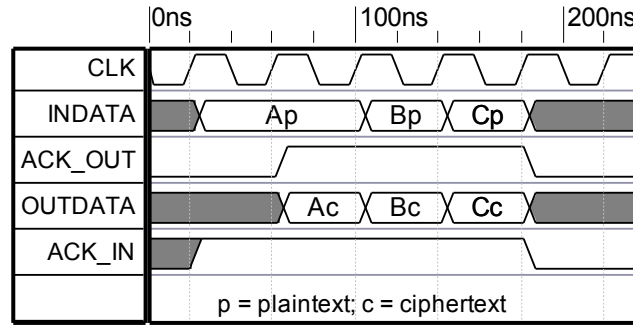


Figure 6.8.: Signal timing for the crypt IP core

the next Section 6.4.3. Figure 6.8 displays the signal timing which is described by the **transfer** block in Figure 6.7 with *INIT* := 1.

During normal operation, the **transfer** block applies a plaintext data item to port *INDATA* and sets *ACK_IN* to start encryption. After receiving *ACK_OUT*, the ciphertext is sampled from *OUTDATA*, and *ACK_IN* is reset. Should *ACK_IN* not be received within 16 clock cycles or *INIT* be reset during that time, the control flow branches to the **exception** block and waits for *INIT* to be re-enabled again to start over. The sequence in Figure 6.9 is defined in the CoMAP specification for the port *INDATA*.

```

; data
sequence
  bigendian: 32 bit signed
end sequence

```

Figure 6.9.: CoMAP sequence definition for port *INDATA*

From the behavior and sequence descriptions, the **crypt** function in the example of Section 6.1 can be generated. The function terminates after encrypting one data word (note the omission in Figure 6.9 of the optional **repeat** statement in contrast to the sequence in Figure 6.4) from the memory pointed to by *plaintext* and delivering it to **ciphertext*:

```
crypt(int *plaintext, int *ciphertext)
```

For both primitives and monoliths, rules for mapping C function parameters to HW ports are required, which are described in the next section.

6.4.3. Mapping C Function Parameters

This section describes the details of mapping hardware interfaces to C function parameters. The following rules apply whenever a primitive or monolith is generated:

- For each CoMAP sequence in a port declaration (see Section 5.4.2), a C function parameter is created. Alternatively, if no sequence is defined for a port, a single

function parameter with the full port width is created (cf. Section 6.3.2). The parameters are enumerated in the order that they appear in their enclosing interface declarations in the CoMAP component structure. Thus, the generated C function signatures are reliably reproduced. In case of a monolith, the same enumeration scheme is used for the **proc** statement in a **behavior** declaration (see Figure 6.7), which in turn maps between the ports or sequences (evaluated as positional parameters) and their logical names used in the UCODE.

- The function parameters are always pointers to the type that is defined in the respective sequence or port description. A best match approach is employed for mapping scalar hardware data to C data types. The n least significant bits of the next larger C type represent a hardware scalar of width n . If a sequence comprises multiple body elements, they are combined to a C **struct**, again applying the best match approach to each struct member. The function return type is always **void**, thus nothing is returned. If a function requires to return a value to the caller, one of its parameters can be used as the return channel, which has to be fed by the UCODE behavior.
- If a port either acts as slave on a bus (**access: slave**, see Section 5.4.2) or lacks an address output in bus master mode (**access: master** or **masterslave**), then a stream engine is automatically generated which transfers data between memory and read port or vice versa on a write port. The stream engine is configured for the data traffic requirements that are described in the **traffic** block (see Section 5.4.2) of the corresponding port. With a stream engine, addresses for an optional address input on such port have to be provided via UCODE.

The next section shows the integration of PaCIFIC's mechanisms and concepts into a research C compiler.

6.5. Compiler Integration

The PaCIFIC approach requires additional processing to be inserted into a C-to-HW compiler. These extra steps (compiler passes) access the PaCIFIC descriptions to find idiomatic HW function calls in the C source program. As first practical realization, the Compiler for Adaptive Systems (Comrade) [GäKo08], which is under development at the Embedded Systems and Applications group (TU Darmstadt) and the Department of Integrated Circuit Design (TU Braunschweig), will act as the host compiler. At the time of writing, Comrade is reworked in many core aspects and transferred to a modern compiler framework. The details described in the next Section 6.5.1 reflect the current stable version of Comrade. Nevertheless, PaCIFIC is designed to operate on the new version as well. PaCIFIC enables Comrade to access and integrate IP cores which are too complex to be generated efficiently just from a SW description (described in Section 6.5.2).

6.5.1. Comrade Compiler System

The Comrade compiler system (see Figure 6.10) aims at automatic HW/SW partitioning of ANSI C programs based on dynamic profiling data. Long running parts of the program (often loop nests) are then scheduled for hardware execution if possible. Comrade composes the heavily pipelined hardware datapath from simple arithmetic operators, which are generated from a parameterized module library (shown as **ModLib** in Figure 6.10). The current stable version of the compiler itself is based on the SUIF2 compiler framework [Lam99], extended by a Control Memory Data Flow Graph (CMDFG) [GäKo08] based on the Static Single Assignment (SSA) form [CFRW91]. After subjecting the C code to target-independent high-level optimizations [Appe98], the resulting intermediate representation (IR) is converted to CMDFG form, which is the basis for the following compiler passes.

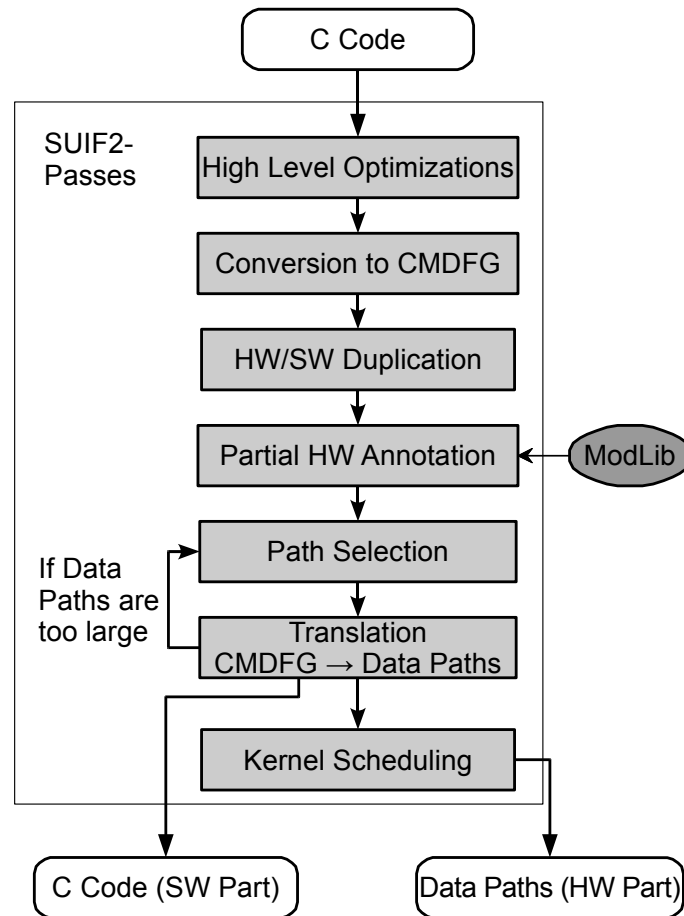


Figure 6.10.: Comrade compiler flow

The HW/SW partitioning step in the compiler operates mostly on a high IR level: Loops are still intact and have not been dismantled into IF/GOTO constructs. Only later, when actually generating HW datapaths, will the IR level be lowered. All atomic operations, such as arithmetic and logic primitives, however, are already annotated

with the characteristics of their possible HW realizations (computation time and area requirements). Due to their atomic nature, these operations and their associated HW characteristics will not be affected by later transformations such as loop unrolling or switch statement dismantling. Thus, the HW data annotation step can be performed this early in the compile flow.

For each atomic operator, Comrade retrieves estimated HW characteristics from the module library to determine the cost of implementing program operations in HW. These characteristics combined with execution profiling data form the base for selecting regions in the control flow graph (CFG) for HW implementation. Currently, Comrade is improved to use advanced dynamic profiling techniques such as *path profiling* [BaLa96] and *whole program profiling* [Laru99] employing address and data traces.

Possible HW blocks are then assembled from the candidate regions in a path-by-path fashion from performance-critical loop nests. Note that the selections which are made by this algorithm are just HW candidates that will be evaluated in greater detail in later partitioning steps not shown here. Unsuitable paths such as those containing I/O or HW-infeasible operations (e.g., calls to complex library functions) are eliminated from consideration. In some cases, the HW-selectability of a path is dependent on the input data (e.g., an illegal value causes the output of an error message). For these cases, the path is duplicated in both HW and SW (analogous to GarpCC [CaHW00]). The HW-infeasible exception-handling code is realized only in SW. At the appropriate HW location, a HW-SW execution switch to the SW version is inserted, allowing a run-time decision between HW and SW execution. Furthermore, by keeping the SW duplicate, later compiler passes can easily unselect a HW candidate should it prove too costly and switch back to its SW version. E.g., area constraints can limit the possible choices for HW execution to just the innermost loop of a loop nest. Later passes can then select the best choice from this spectrum of possible partitionings.

Additionally, interface blocks are inserted at all HW-SW and SW-HW interfaces to allow data transfer of the live variables between GPP and hardware accelerator. The HW paths that have been actually selected are then realized as datapaths on the target hardware accelerator (see Section 4.3), a process that will not be described further here. They are ranked according to the performance gain relative to the SW realization, which results primarily from the number of parallel operators in the HW datapath. Only the best HW candidate regions according to this metric will actually be realized in HW, the rest will use their SW variants.

The decision to create a HW candidate from a program region by duplication also depends on the presence or absence of SW function calls within the region. SW function calls (in contrast to HW IP “function” calls) cannot be realized directly in HW. However, they can be eliminated by inlining their code into the calling function (leading to a larger required HW area, though). The choice whether to inline can be made based on profiling data and be limited to time-critical functions.

The next section describes the PaCIFIC extensions to the standard compiler described above, which enable automatic IP core integration.

6.5.2. PaCIFIC Extension to Comrade

In the first PaCIFIC compiler pass, which is planned to be inserted into the new Comrade version after the high level optimizations (see previous section), the IR of the C program [KKGR03, GäKo08] is scanned for IP cores. A function is recognized as IP core if

1. the function is explicitly declared as HW either by a separate definition file or markings in the source code and
2. an IP core with a signature matching the function is found by PaCIFIC in the CoMAP repository.

These calls are flagged to be excluded from the existing C-to-HW compilation passes. Thus, other statements in the direct vicinity of the IP call are still processed normally. The IR node for the basic block that holds the IP calls is thus split between SW execution and synthesized HW datapaths on the one hand and IP core execution on the other hand (see Figure 6.11).

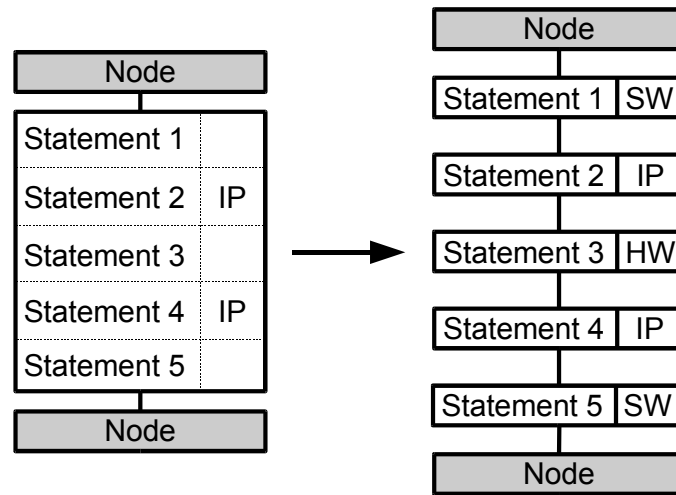


Figure 6.11.: Control flow before and after split

After the non-IP core code has been partitioned into HW and SW parts, the IR is searched for HW pipelines. These are assembled both from HW-compiled datapaths as well as from the IP calls which are adjacent in the control flow (see Figure 6.12). The pipelines will be re-inserted into the existing compile flow after the path selection.

For well-behaved loops (constant per-iteration increment of induction variable, fixed loop bounds), equivalent stream engines can be automatically generated. If the combined HW node is the only node in a loop body and no data dependencies exist outside of the loop, the entire loop is thus transformed into a corresponding streaming HW datapath (removing explicit induction variable arithmetic).

Under certain conditions, the loop can be parallelized (e.g., by unrolling). In this case, and subject to resource limitations, multiple independent instances of an IP core may work on their respective sub-streams simultaneously.

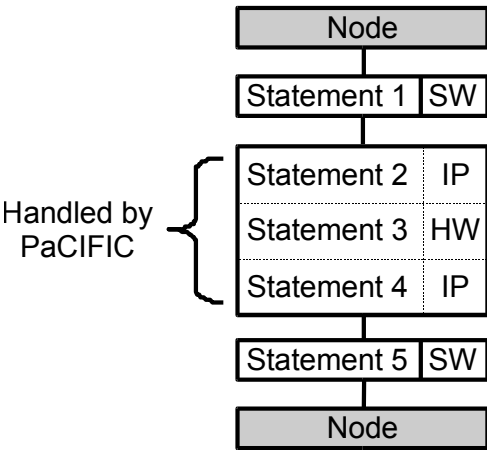


Figure 6.12.: Pipeline of combined HW nodes

The next section will present an example which demonstrates the interplay of the new compiler passes.

6.5.3. Real World Example: Automatic FFT IP Core Integration

To demonstrate the new Comrade compile flow for IP core integration, the Xilinx High-Performance 16-Point Complex FFT/IFFT [Xili01] of the Core Generator suite is coupled to an ANSI C program (shown in Figure 6.13) applying the PaCIFIC algorithms manually, since the automatic tool flow is not yet fully operational.

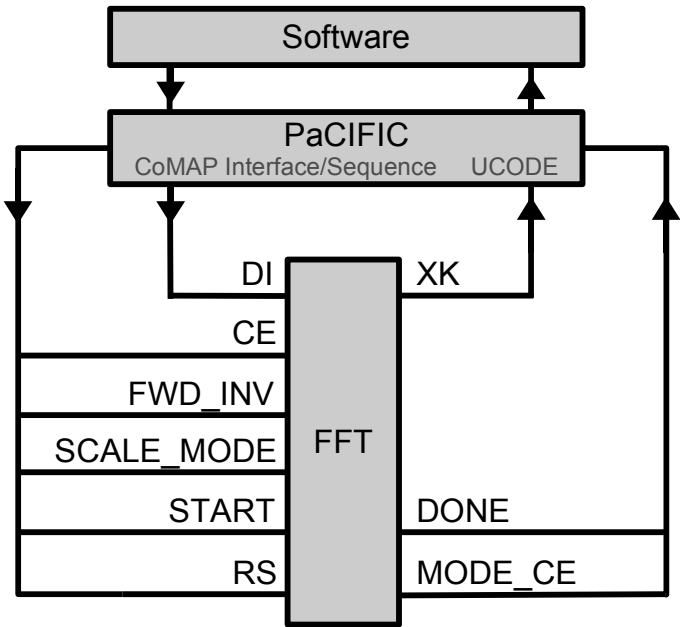


Figure 6.13.: FFT pipeline controlled by PaCIFIC

```

int main(int argc, char* argv[]) {
    FILE* infile, * outfile;
    int* dram_in, * dram_out;
    #pragma HARDWARE vfft16(int*, int*)

    infile = fopen("time.dat", "r");
    outfile = fopen("freqspec.dat", "w");
    dram_in = calloc(16384, sizeof(int));
    dram_out = calloc(16384, sizeof(int));
    fread (dram_in, sizeof(int), 16384, infile);
    vfft16(dram_in, dram_out); /* HW function call */
    fwrite(dram_out, sizeof(int), 16384, outfile);
    ...
}

```

Figure 6.14.: C program calling FFT hardware accelerator

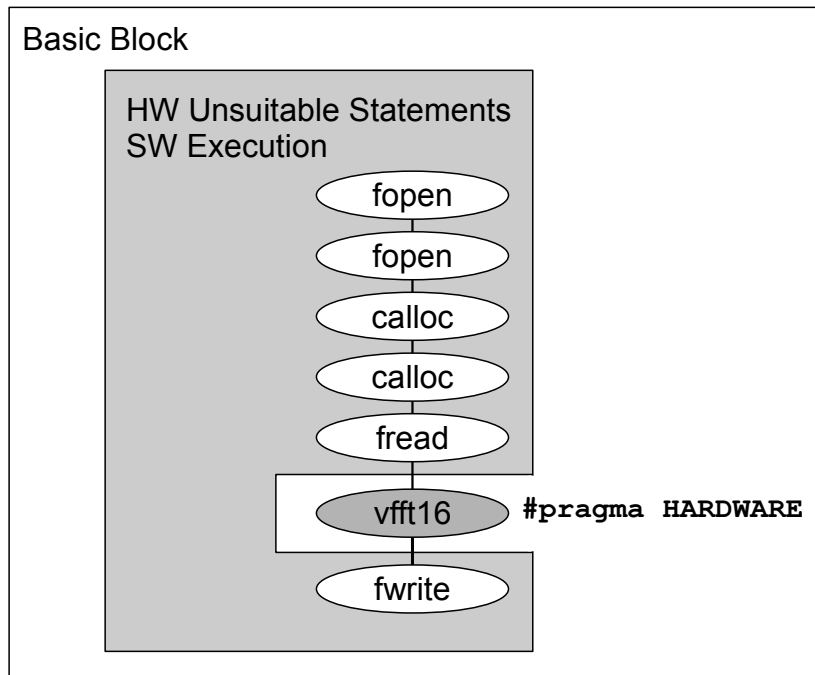


Figure 6.15.: Intermediate representation (IR) of FFT application

The FFT expects data to be continuously streamed to its input buses as well as from its outputs. For simplicity, the 16 bit real and imaginary buses are combined into 32 bit buses carrying complex numbers. The output data is available after an initial latency of 82 cycles. The C program reads the source data from a file into the DRAM, calls the FFT hardware accelerator, and finally writes the result back to disk (see Figure 6.14).

```

; fft16
behavior fft16
proc vfft16(time, freq) ; forward fft

; initialize
posedge CE=1 SCALE_MODE=0 FWD_INV=1 START=1
posedge START=0

; pipeline steady-state starts here
; transfer 256*16 samples = 16KiB data per invocation
start 256

; wait for acceptance of first FFT block
continue timeout: 16 MODE_CE=1

; write 16 time domain samples
transfer 16 DI
    posedge DI=time
end transfer

; fork control flow for pipelining
restart

; wait for transformed data
continue timeout: 82 DONE=1

; read 16 frequency domain samples
transfer 16 XK
    posedge freq=XK
end transfer

; reset IP core
exception
posedge RS=1
posedge RS=0

; execution terminates here
end behavior

```

Figure 6.16.: UCODE for behavior “fft16”

```

component FFT
  version: 1.0
  behavior
    ; See Figure 6.16
    ...
  end behavior
  interface general
    port DI
      transaction: data
      direction: input
      width: 32
      ; Input 16 time domain samples
      sequence repeat: 16
      bigendian: 32 bit unsigned
    end sequence
    traffic linear
      blocksize: 64
      burstiness: 15% burstsize: 16Ki
    end traffic
  end port
  port XK
      transaction: data
      direction: output
      width: 32
      ; Output 16 frequency domain samples
      sequence repeat: 16
      bigendian: 32 bit unsigned
    end sequence
    traffic linear
      blocksize: 64
      burstiness: 15% burstsize: 16Ki
    end traffic
  end port
  ...
end interface
end component

```

Figure 6.17.: CoMAP description for ports DI and XK

Figure 6.16 depicts the UCODE for wrapping the FFT IP core. After programming the operating mode, it accepts a 16-sample block of time-domain data via port DI (see also Figure 6.13). After the end of the computation is indicated, 16 frequency-domain samples can be unloaded from the core via port XK. In a pipelined fashion (UCODE **start/restart**, see Section 6.3.2), the next set of time-domain data (256 sets total) can be provided to the core when it becomes available again. Note that all FFT control signals (e.g., **CE**, **SCALE_MODE**, **START**, **DONE**, etc.) are driven and monitored by the UCODE logic, which is synthesized to hardware and completely hidden from the C program.

The PaCIFIC algorithms are now applied in compile order, which corresponds to their respective locations between the existing Comrade passes (described in previous Section 6.5). After the high level optimizations, the IR (see Figure 6.15) of the C program (Figure 6.14) is scanned for IP cores. Since the function call to **vfft16** has been marked as **HARDWARE** (first condition from previous section) in the C source code (or in a separate definition file, not shown here), Comrade/PaCIFIC searches the CoMAP repository for an IP core with a matching signature (second condition). As described in Section 6.4.2 above, the signature of a monolith is defined by the **proc** statement in the UCODE description (Figure 6.16). The FFT core is found (see corresponding CoMAP repository entry shown in Figure 6.17), and its SW function call is flagged to be excluded from Comrade's normal C to HW compilation mechanisms. The IR node for the basic block that holds the call to **vfft16** is subsequently split between Comrade's normal SW execution and HW datapath sections on the one hand, and the **vfft16** section on the other hand (see Figure 6.18 a and b).

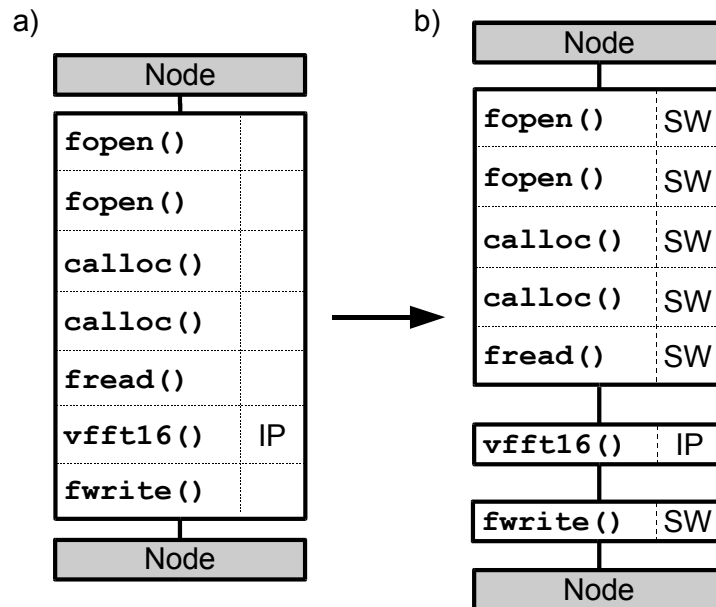


Figure 6.18.: IR node containing basic block before and after split

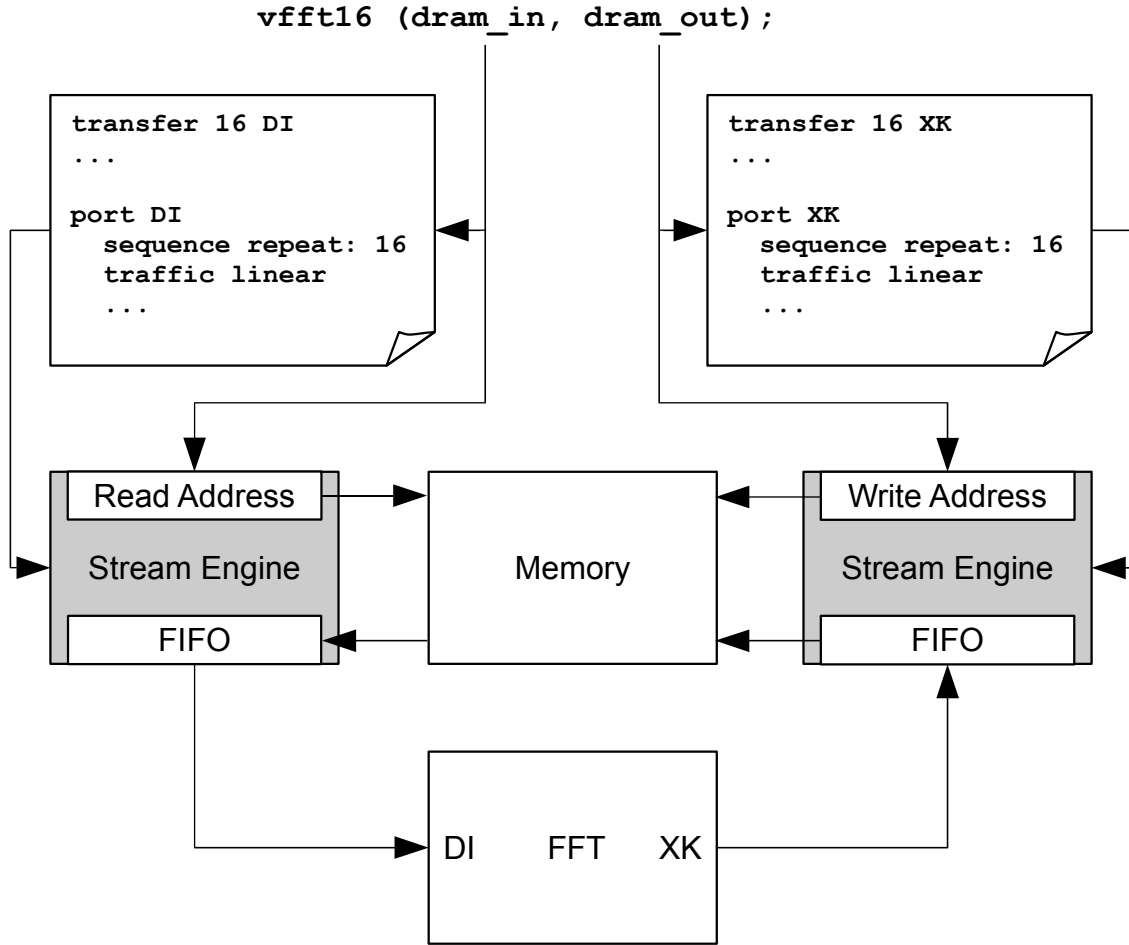


Figure 6.19.: Stream engines operating from pointers source and sink memory areas

After the non-IP core code has been selected for pure SW execution by Comrade's usual mechanisms (all remaining C statements are not suitable for HW), the IR is scanned for IP core pipelines. Since there are no HW datapaths synthesized by Comrade, the single pipeline consists of the FFT IP core only. In a final step, this pipeline is re-inserted into the normal compile flow after the path selection (Figure 6.18 b).

The data flow between the SW and HW parts of the FFT application is established using the two C pointers `dram_in` and `dram_out`, which refer to memory areas that hold the source and result data for the FFT IP core (see Figure 6.19), respectively. The data for the FFT logic is sourced and sunk by two stream engines which access the memory in bus master mode using the pointers as addresses. To this end, either AISLE (described in Section 4.6.2) or PHASE/V (Section 4.6.3) can be used to provide for compatible pointers that are freely interchangeable between HW and SW. The stream engines are dimensioned using CoMAP's traffic specification (see Figure 6.17) for the ports DI and XK, which yields a FIFO capacity of 256x32 bit each. The naive approach without PaCIFIC would require a manual set-up of the stream engines and the control signals for the FFT. Instead, all of this is wrapped by PaCIFIC into a single function call.

As described in Section 6.4.3, both parameters of the monolith generated from the `vfft16` C function call (see Figure 6.14) are mapped by position employing the correspondingly named UCODE `proc` statement (Figure 6.7), which is associated with the CoMAP behavior description for the FFT IP core. The first C function parameter `dram_in` is thus mapped to the first UCODE `transfer` statement, which also corresponds to the first (and only) CoMAP sequence for the port `Dl` (see Figure 6.17, also shown in Figure 6.19). The second parameter `dram_out` is mapped to the second UCODE `transfer` statement which corresponds to port `XK`. However, `dram_out` is also mapped to a sequence: Since there is no further sequence defined for port `Dl`, the second sequence in the CoMAP interface is used as mapping target, which is consistently defined for port `XK`.

Both transfer statements and sequences indicate that 16 combined samples are transferred per invocation of the FFT core. In addition, the pipelining mechanism provided by `start restart` iterates over 256 invocations of the core (cf. Figure 6.16). Hence, a single call to the `vfft16` C function results in a total of 4096 samples (i.e. 16KiB of data, cf. C code shown in Figure 6.14) being processed.

The effectiveness of the combined CoMAP and PaCIFIC approach will be demonstrated in Section 8.2 by giving implementation results for this FFT example.

6.6. Chapter Summary

This chapter described a method for integrating hardware intellectual property (IP) cores from a software programming language, thus broadening the ability of hardware/software compilers such as Comrade to build large hardware/software systems which are required to efficiently construct a high-performance Reconfigurable Computing Platform, the main objective of this thesis. To this end, a system and interface communication model was developed based on the evaluation of many commercial IP cores, which also exploits the detailed hardware core and interface definition features of CoMAP. For the hardware interface, two classes of properties were identified to define static as well as dynamic interface behavior, the latter based upon an existing synthesizable protocol language. Two corresponding interface types were then designed for the software side, which provide for automatic simple interface generation and encapsulation of complex IP core functions in a single function call respectively, while maintaining a natural C programming style. The interplay of hardware and software as well as the data exchange between them adhere to the execution model and its underlying architectures developed earlier in this work. Furthermore, the realization of this combined hardware/software approach was demonstrated using Comrade as the host hardware/software compiler to depict the additional compiler passes for the new PaCIFIC mechanisms. Finally, a real world example was shown, which uses conventional C language to seamlessly integrate a standard IP core with the Reconfigurable Computing Platform that is complemented by the other parts of this work.

7. Speculative Memory System

Why is speculative program execution especially effective in the context of hardware-accelerated computing? Does an efficient speculative memory system always have to be costly?

In a typical SW program, about 20% of the instructions are memory accesses that need up to 100x as much execution time as a simple non-memory instruction [KaYe05, Chapter 14]. Such long execution times originate from memory latencies which can only partially be hidden by cache memory hierarchies. This attempt by itself completely fails whenever bulk data is to be fetched, processed, and written back sequentially, a type of streaming data that is typical for signal processing applications, which strongly benefit from hardware acceleration.

Hence, there is considerable potential for accelerating program execution by *memory speculation*. In contrast to the general case of GPP-based speculative program execution (discussed in the next section), where all types of program instructions including operand data and addresses are speculated, memory speculation starts up to as many memory accesses in parallel as memory ports are available. Although the early start of accesses is similar to prefetching, memory speculation goes beyond mere latency hiding by prioritizing those accesses that have a high probability to be required by the program trace being actually executed. Thus, the scheduling of less important accesses is delayed until otherwise idle memory bandwidth becomes available. In the extreme case, those accesses are never executed or completed at all: As soon as the program flow resolves an access as unnecessary, it is cancelled, thus effectively saving bandwidth.

To this end, it must be determined which accesses can be executed independently (and hence in parallel or even in reverse order) of others. Although statically scheduled speculation is predetermined at compile/synthesis time, and is thus easier to implement than dynamic speculation scheduled at runtime, it is hard to prove independence of memory accesses in advance from pure program code without the actual data flow [KaYe05, Chapter 14]. Hence, dynamic speculation can be employed in all cases when memory dependencies cannot be (dis)proved statically.

This chapter shows the effectiveness and efficiency of pure dynamic speculation without any static analysis. A significant performance gain is achieved at low implementation cost (e.g., without complex load-store queues and speculation controllers). To this end, a lightweight speculative memory system was developed and implemented using non-destructive speculation techniques. The performance gain is substantiated with benchmark results in Section 8.3.

The next section presents the general concept of speculation, which the requirements for a speculative memory system are elaborated from in Section 7.2. Then the implementation

details are described (Section 7.3), followed by the integration of the speculative memory system with the other parts of the Reconfigurable Computing Platform, especially FastLane+ (described in Section 4.5), MARC (Sections 4.5.1 and 7.3.1 below), and the CoMAP platform management (Chapter 5).

7.1. Speculative Program Execution

The naive approach for program execution starts the execution of every instruction only when all conditional expressions it may depend on are evaluated as true, and all input operand data from previously completed instructions is available. In a single-threaded scalar machine this strictly sequential execution model is no limiting factor for program execution speed, since the single arithmetic logic unit (ALU) is only capable of completing one computation at a time. Hence, a following instruction cannot start at an earlier time, since the single compute resource of the GPP is still busy with the previous instruction. However, modern GPPs [Inte09] [HePa06, Chapters 3, 4] use a multi-threaded superscalar execution scheme, which processes multiple instructions in parallel. The degree of parallelization varies dynamically and depends on the type of instructions being executed, since there are different numbers of instances per type of execution unit (e.g., ALU, fetch-decode, memory/register buses), well balanced for the instruction mix found in common programs.

As the logical execution flow of a program remains strictly sequential for most SW programming languages and paradigms, it is necessary to automatically identify sequential instructions in a program that can be executed in parallel to exploit such superscalar processor architectures. In the conventional, non-speculative execution scheme described above, a data or control dependency between instructions must be resolved prior to execution of the dependent instruction. However, *speculative* program execution does not wait for all prerequisites of an instruction to be fulfilled, the instruction is started as soon as resources in the processor pipeline become available. To this end, the required prerequisites are *predicted* until their calculation can eventually be completed. If the prediction has been correct, execution can continue normally, otherwise the execution of all instructions that depend on the mispredicted prerequisites has to be squashed and restarted with amended prerequisites.

The following sections describe the different types of speculation (i.e., *what* can be speculated), the peculiarities of memory-based speculation, and what impact the differences in the execution models of GPPs and ACSs have on speculation.

7.1.1. Speculation Types

The term *control speculation* generally [HePa06, Chapter 2.6] denotes the start of execution despite yet unavailable conditional prerequisites, e.g., whether a portion of code should be executed at all (shown in Figure 7.1). If sufficient resources are available, multiple or all alternative program branches can be executed in parallel, eventually keeping the correct branch when the condition has been evaluated. *Branch prediction* is a special

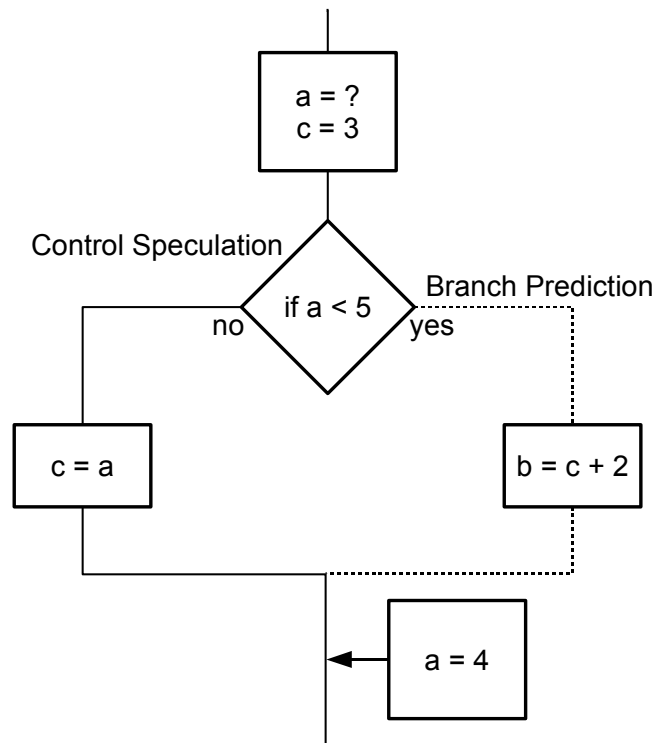


Figure 7.1.: Program flow with control speculation

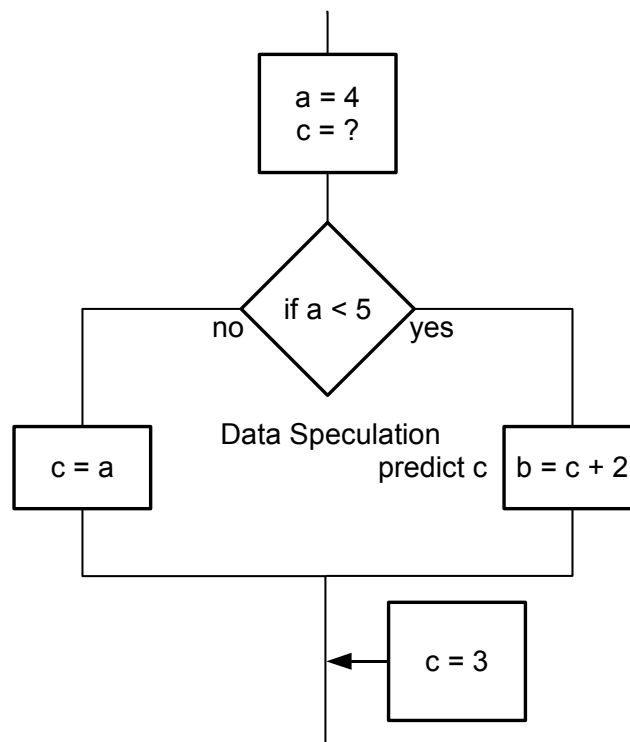


Figure 7.2.: Program flow with data speculation

case of control speculation where the branch condition is predicted and only the likeliest branch is executed.

In contrast, *data prediction/speculation* (see Figure 7.2) is used if operand data is not available in time for an instruction that performs, e.g., an arithmetic calculation, which would otherwise have to be delayed until all operands finally have been calculated. To this end, operand data is estimated using predictors of varying complexity, ranging from simple constant/last value predictors to sophisticated pattern history-based schemes. Conceptually, read data can be predicted as well as write data. Note that control and data speculation can also be combined (Figure 7.3, a is both control and data variable here).

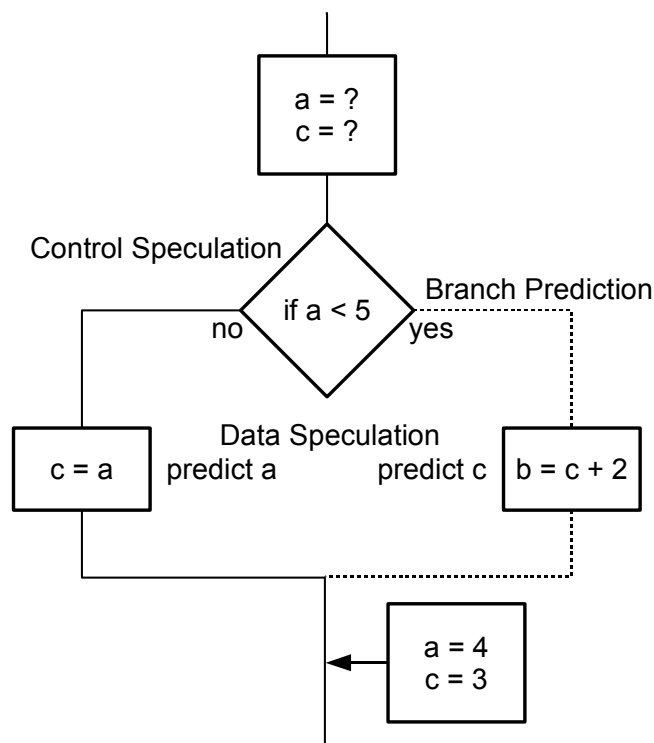


Figure 7.3.: Program flow with combined control and data speculation

Despite its potential for acceleration [KaYe05, Chapter 9], data value prediction is not implemented in current processor designs. However, the HP PA-8000 series [ShLi05, Chapter 8.3.3], Intel Itanium/-2, and PowerPC 620 include HW support for data *dependence* speculation [KaYe05, Chapter 14]. Here, instead of the operand values themselves, it is speculated whether an operand depends on other operands (e.g., by memory reference). Instructions that would have to be executed sequentially due to unresolved operand dependencies are now calculated in parallel or out-of-order. If operands are later detected to be dependent, the affected instructions are squashed and re-executed sequentially with amended operands.

Address prediction/speculation is a special case of data speculation. Here, the aim is to predict the addresses for memory accesses, which enables the memory system to

prefetch contents before they are actually needed, e.g., cachelines containing read data or receiving write data. With prefetching, the memory latency can completely or partially be hidden from the execution pipeline, which would otherwise have to be stalled until the memory access is served. Hence, the efficiency of address speculation is closely related to the memory system layout, the characteristics of both should be carefully tuned to each other.

A classification and evaluation of predictor schemes for all types of speculations can be found in [Mart07].

7.1.2. Memory-based Speculation

The concepts of control, data, and address speculation are unaffected by the type of data storage, it may be supplied from external memory, registers, or the internal datapath itself. However from a memory system perspective, speculation is obviously only relevant if a speculated portion of code contains memory accesses. If such accesses are situated in a control-speculated block of code, they may be executed speculatively, although their execution would not be legal by program logic in the unspeculated case. Hence, the memory system (or, alternatively, an other instance that has coherent knowledge of all proceeding speculative accesses) must cater for preliminary memory transfers that can be nullified later. Thus, whenever a misspeculated conditional branch is squashed, the corresponding memory accesses have to be *cancelled*. Simultaneously, the correctly speculated surviving branch must be *committed*. Likewise, preliminary speculated data must also be committed when it becomes eventually valid (possibly after multiple preliminary updates). A *write* commit means that the access is taken (speculation hit) and its data is final. Analogously, a *read* commit means that the reader now expects the final unspeculated data, which is guaranteed to be correct. In contrast, a read or write cancel resets the memory system to a state as if the speculative accesses had never occurred.

To guarantee correct program execution, all memory accesses must be logically completed in *program order*. However, due to speculation, an access may be issued earlier than an other access preceding it in program order. Such speculative violation of program order can be classified into three groups of *memory hazards*:

- **Read after write (RAW) hazard (true dependency)**

Data is read which has been written by a preceding write according to program order (see Figure 7.4). If the read is speculatively issued before the write, it is not known whether the write uses the same address as the read. Hence, the read data may change due to its dependency on the unfinished write, and the read cannot commit before the write. If the write is control-speculated, or its address or data are unknown or speculated, the read must not commit until the write is committed or cancelled.

- **Write after read (WAR) hazard (anti dependency)**

Data is read which is overwritten by a subsequent write according to program order

(see Figure 7.5). If the write is speculatively issued before the read, it is not known whether the read uses the same address as the write. Hence, the original data stored at the write address must be preserved from early overwriting.

- **Write after write (WAW) hazard (output dependency)**

Data is written which is overwritten by a subsequent write according to program order (see Figure 7.6). If the second write is speculatively issued before the first (non-speculative) write, it is not known whether both use the same address. Hence, the first write can accidentally overwrite the second, which must be avoided by preserving both data values for later resolution of the conflict.

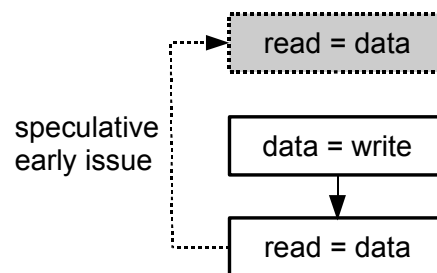


Figure 7.4.: Read after write (RAW) hazard

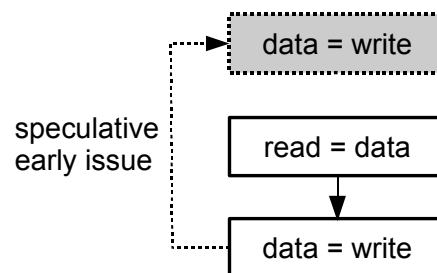


Figure 7.5.: Write-after-read (WAR) hazard

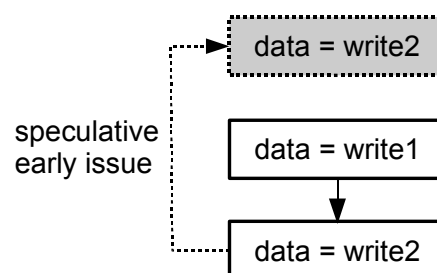


Figure 7.6.: Write-after-write (WAW) hazard

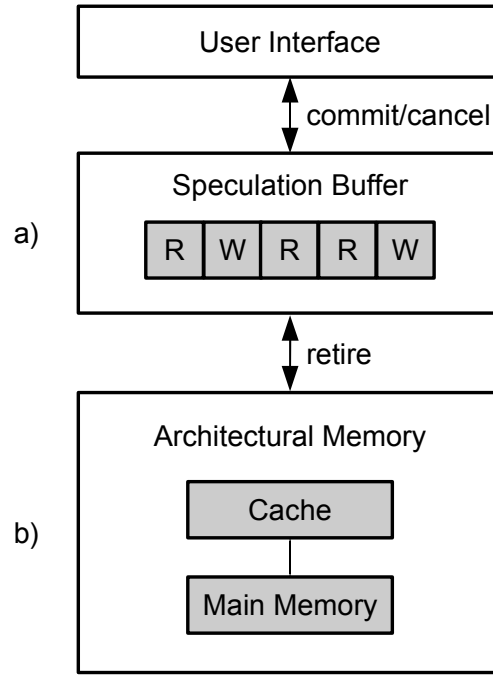


Figure 7.7.: Memory hierarchy for memory-based speculation

To manage these hazards, a speculative memory system needs an additional mechanism (often referred to as *memory disambiguation* in the GPP context [KaYe05, Chapter 14.1.3]), which is sometimes integrated with the cache. In this *multi-version cache* [GVSS98], the cachelines are used to also store the speculative versions of all memory operations. A more common approach, which can also be combined with multi-version caches [HaWO98], introduces an additional level of memory hierarchy above the common cache hierarchy: The *speculation buffer* (Figure 7.7a) contains all speculative accesses that are still in flight. Complementarily, the term *architectural memory* (see Figure 7.7b) denotes the underlying conventional (cache) memory hierarchy of the system that by definition only holds unspeculated or committed data. Finally, *retire* means any eviction of a memory access from the speculation buffer, regardless of whether its data is discarded (cancel) or written to architectural memory (commit).

Obviously, read operations cannot alter memory contents, they are *non-destructive*. Hence, no data needs to be preserved from accidental speculative overwriting. In contrast, write operations are *destructive*. Since write operations alter the memory contents, the original values that are stored in the memory before writing must be preserved for later restoration in case of a failed speculation. In other words, the new speculative data must not be retired to architectural memory until the outcome of the speculation is known.

Additionally, each value of *consecutive* speculative writes to the same memory address (WAW) has to be stored separately, since each write operation may be cancelled at a later time, and the final value for every memory address must be remembered eventually (see Figure 7.8). Hence, a write must not retire until the addresses of all previous writes are known to be different from that write's address, or all previous writes have retired.

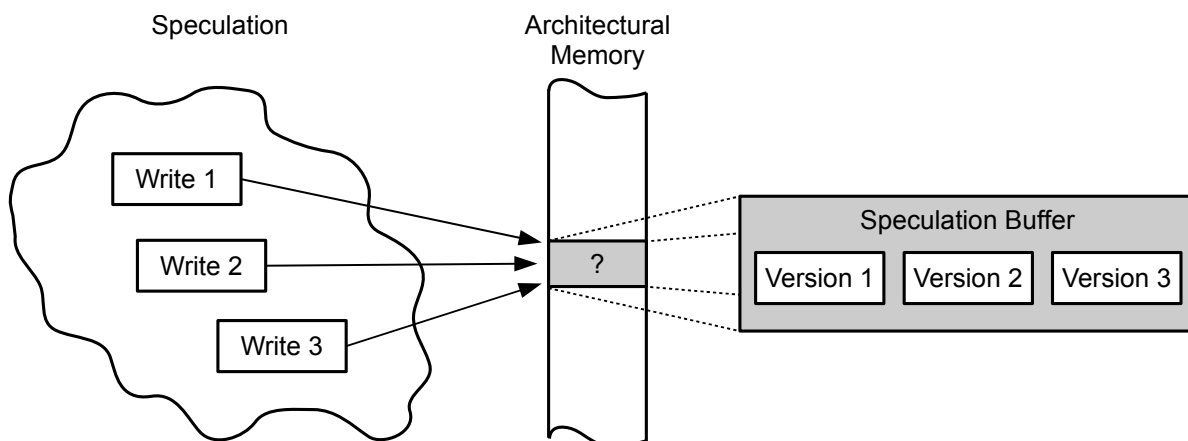


Figure 7.8.: Speculative writes generate multiple versions of the same memory address

If write data is speculated, the memory system needs to handle *write updates*. Write data may be updated multiple times per access due to failed write data speculations. Such mechanism informs all dependent read operations (RAW) of updated speculative or eventually rectified data, the latter when the write is committed. The data is subsequently forwarded to the read operations (*store-to-load forwarding*), which may choose to restart their associated calculations with the new data if required. To this end, a read must not commit until the addresses of all previous writes are known to be different from the read address. Note that read operations also have to be held in the speculation buffer until retirement, since their values may depend on previous speculative writes.

The simultaneous presence of multiple versions, which can only be created by destructive operations, also implies that the view of the memory is no longer *coherent* for all operations. Memory operations can only depend on each other if they are in the same (speculated) control branch. Hence, if data has been speculatively modified by a write, the update must only be visible to dependent operations in the same branch (see **coherent branches** and the different values for y in Figure 7.9). In contrast, pure non-destructive operations never induce memory incoherency, as there is no need to create multiple versions.

Beyond merely resolving a memory hazard when it has already occurred, it is worthwhile to predict whether an address or data dependency exists between two or more accesses in the first place [KaYe05, Chapter 14]. If an access is predicted to likely depend on a previous access according to program order, it is less desirable to speculatively execute the dependent access first (see Figures 7.4–7.6) due to the expected misspeculation. From a memory system perspective, *dependency prediction* is closely related to address prediction, since memory accesses can only interdepend if they share the same address. Likewise, *access prediction* attempts to predict whether a speculative memory access is likely to commit. Accesses with a low commit probability are less desirable for early speculative execution, since memory bandwidth would be wasted, and accesses with a higher commit probability would be unnecessarily deferred. Access prediction in the memory domain is related to control prediction: Memory accesses located in a control-speculated program branch can share the same control predictor.

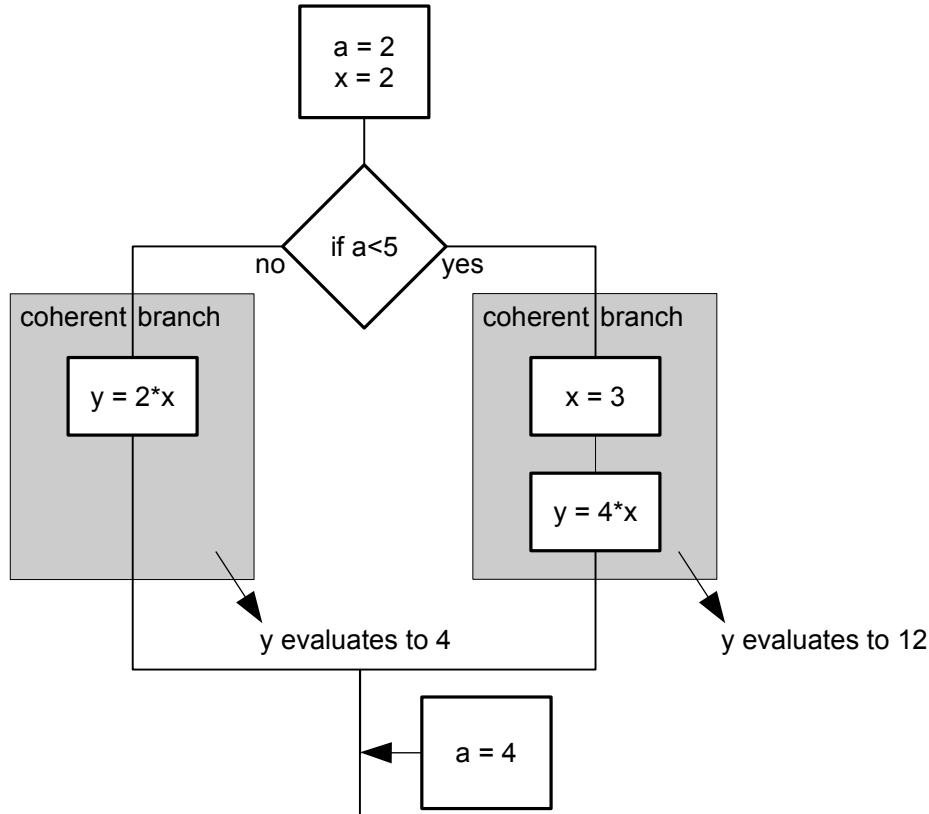


Figure 7.9.: Memory updates are only visible in the same coherent control branch

Using both dependency and access prediction, the precious memory bandwidth (see Chapter 4.5) is exploited efficiently only by the most promising access(es). Note that the prediction of memory access dependencies is not necessary under certain circumstances: As long as all memory nodes execute in parallel and squashing as well as re-execution of a partial result at any node impose no time penalty, aggressive (greedy) speculation can be employed without degrading overall performance.

7.1.3. Memory Speculation: GPP vs. ACS Execution Models

Speculative program execution has been explored mostly in the context of SW running on GPP architectures. Pure SW approaches, lacking HW support for speculation management, have to maintain the speculation infrastructure (e.g., buffers holding speculative data, mechanism for tracking, squashing and restarting execution) completely in SW. Thus, such approaches exhibit poor performance due to high speculation management overheads and high misspeculation penalties (the GPP is unavailable for applications during SW speculation management and recovery of misspeculation). The overheads sometimes completely outweigh any possible speculation gains, which compromises all such efforts. Hence, a lot of work has been brought forward to incorporate HW support for speculative execution into GPP(s) and/or memory systems (good introductions

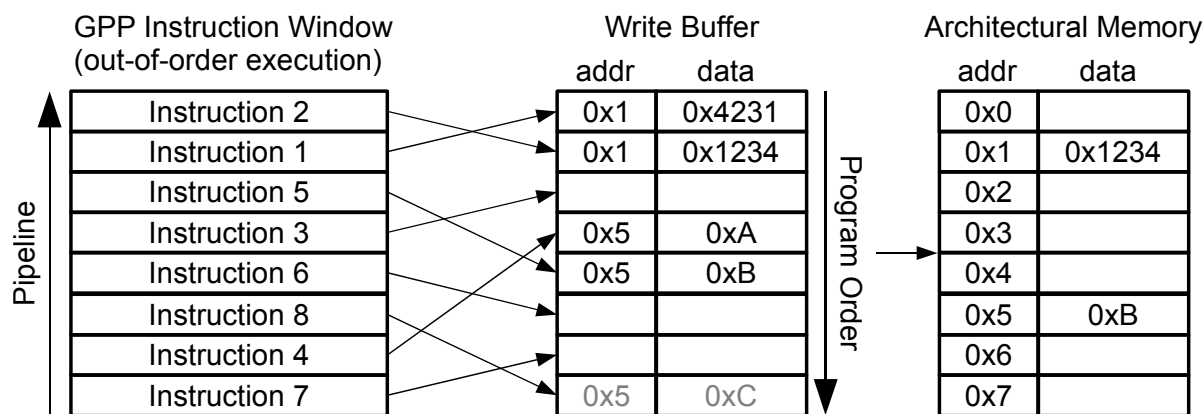


Figure 7.10.: A write buffer disambiguates write operations

can be found in [ShLi05, Chapter 10.4] [KaYe05, MBMH06, SDBM03]). However, the application of the general speculation concepts is not limited to the GPP domain. An ACS using a parallel execution model for the HA (e.g., the Comrade execution model, see Chapter 4) can also benefit from speculative program execution, as will be shown in Section 8.3. However, there are significant differences between the GPP and ACS execution models which have an impact on the speculative memory system.

In a single-threaded execution model, which is common to conventional GPP architectures [ShLi05, Chapter 8], speculation can be exploited by branch prediction, address/data prediction, and out-of-order execution of program instructions [HePa06, Chapter 2]. While the GPP pipeline must cater to the speculatively permuted instruction order and incorrectly speculated control branches, this does not have any consequences for the memory operations as seen by the memory system: Even on superscalar processors, a single GPP-integrated *write buffer* (also known as *store queue* or *reorder buffer*) [HePa06, Chapter 2.6] [ShLi05, Chapters 5.3, 8] is used to reorder, update, or invalidate write instructions (a process known as *memory disambiguation*) before they are passed to the memory system.

The write buffer (see Figure 7.10) is designed to accommodate all write operations that are (speculatively) in flight in the GPP’s instruction window, before they can be retired to architectural memory. The non-destructive read operations (sharing the single memory port with the writes) need not be buffered, but the pipeline must resolve RAW dependencies (e.g., by store-to-load forwarding). With the GPP pipeline executing out-of-order, addresses and values of write instructions are temporarily stored in a data structure (e.g., implemented as a ring buffer) that retains original program order information. A write can be retired and thus be removed from the write buffer when it is cancelled, or when it is committed and the addresses of all preceding writes in the buffer are known to be different. Adhering to this rule, the write buffer arranges all writes in program order. Thus, the correct data value for address 0x1 (shown in Figure 7.10) is written to architectural memory despite the reversed execution order of write Instructions 1 and 2. Note that Instruction 8 is a misspeculated write, which is eventually evicted from the write buffer. Hence, Instruction 5 is the latest valid write to address 0x5, and the value

```

int a, b, c, d, e;
...
if (a < 5)
    d = (a + b) * c;
else
    e = a + b;

```

Figure 7.11.: C program fragment compiled into parallel datapath (Figure 7.12)

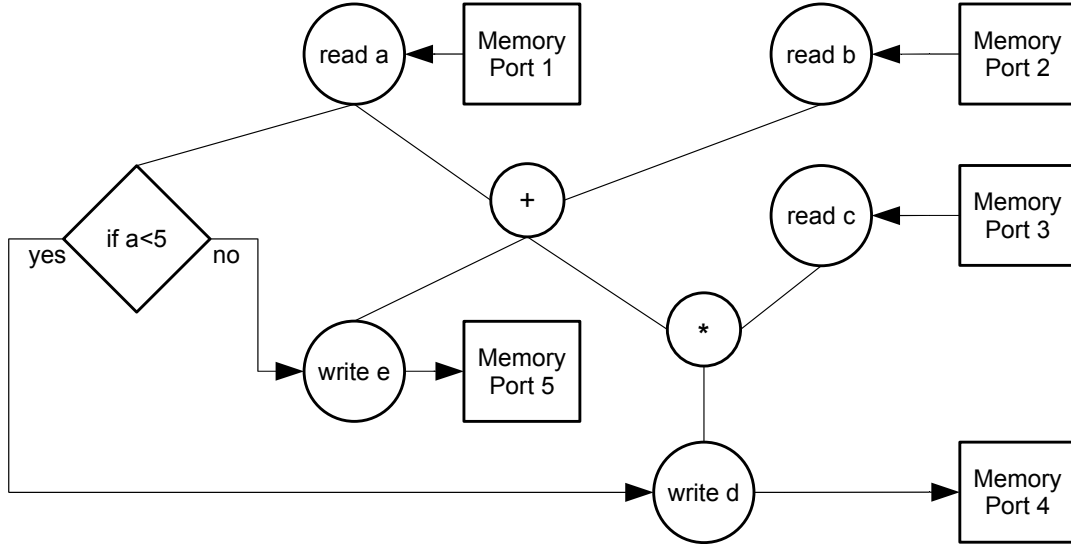


Figure 7.12.: Parallel datapath - each memory operation uses a separate memory port

0xB is written to architectural memory.

While the write buffer is a viable approach that enables speculation for single-threaded machines without adding complexity to the memory system, it does not scale well when moving to multi-threading, many-core architectures. When speculating in thread granularity and assuming that the threads themselves be strictly sequential, thread interdependencies may still lead to speculative hazards. Such dependencies would require a central mechanism to resolve the hazards. Unfortunately, a centralized structure (e.g., a single write buffer) is a bottleneck which memory accesses from all cores must pass, leading to diminished speculation benefits with increasing numbers of cores [ShLi05, Chapter 11.5.3] [KaYe05, Chapter 13.4.2]. An alternative approach could employ a separate write buffer for each core, which would inhibit inter-thread speculation, again affecting speculation gains.

In contrast to the conventional GPP-only architectures, the Comrade execution model enables massively parallel calculations for the HA, both in spatial (e.g., by unrolling) and time (pipelining) domains (see Chapter 4). Here, the parallel datapaths rely on equally parallelized memory ports, which feed and drain data to and from the pipelines. The granularity of the memory ports can be as fine as one port per memory node in the datapath (see Figure 7.12, realizing the C program fragment shown in Figure 7.11).

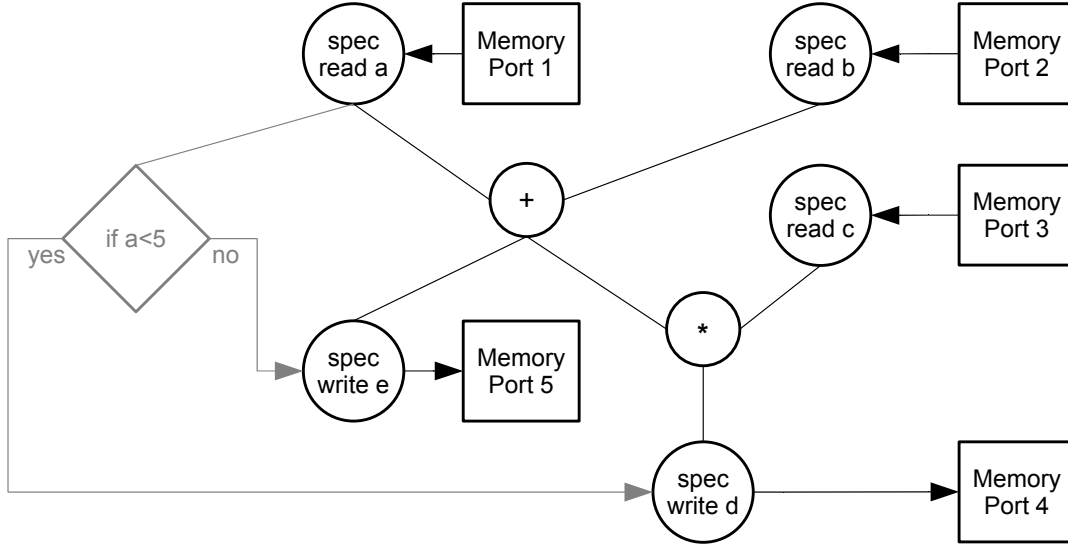


Figure 7.13.: Parallel datapath - speculative execution of memory accesses

While Comrade aims to use as many memory ports as possible to optimally exploit parallelism, such bijective mapping is not mandatory. Multiple operations can share a single memory port if required due to limited HA resources.

Although the Comrade execution model does not use out-of-order execution of memory accesses within a single memory node yet, the instruction-level parallelism (ILP) within the datapaths enables multiple memory operations to run simultaneously, which normally would be sequential by program order (still, all operations must commit in program order). Furthermore, control and data speculation in alternative control blocks (similar to *multipath execution*, a good introduction can be found in [KaYe05, Chapter 6]), as well as speculation in unrolled loop iterations executing in parallel imply the need for speculative memory operations. Notwithstanding future support of *intra*-node out-of-order memory operations, the Comrade execution model can already benefit from memory speculation due to its highly distributed memory access nodes with *inter*-node out-of-order issue times. The large number of memory ports at memory node (= instruction) granularity aggravates the bottleneck of a centralized structure for managing memory hazards (see previous section) and thus mandates a distributed system.

As a solution, this work proposes to move the speculation to the memory system where it is encapsulated as a new memory hierarchy level, separating it from the datapath logic. Thus, the advantages of a distributed and a centralized system are combined by keeping local speculation domains and at the same time leveraging the distributed knowledge for inter-thread, inter-instruction speculation. Memory nodes at instruction-level granularity provide for *independent* and *localized* access, data, and address prediction for all program branches and memory operations (see Figure 7.13). The parallel datapath calculates now *speculatively* the C code fragment shown in Figure 7.11. All reads instantly deliver *predicted* operand data to start dependent calculations before the actual data arrive from memory. Calculations may be squashed and restarted should the data prediction

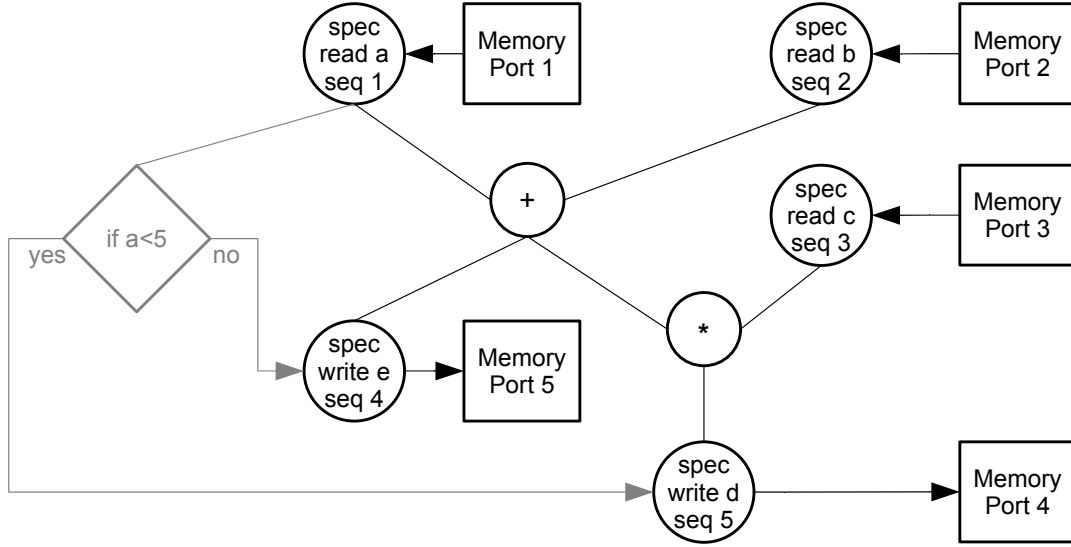


Figure 7.14.: Parallel datapath - sequence numbers represent original program order

be incorrect. The condition ($a < 5$) is disregarded (shown in grey colour) and both accesses **write d** and **write e** are started in parallel. Only when the condition is finally known it is decided which access to commit or cancel. At the same time, statistical data about committed and cancelled operations is collected individually at each memory port. This unique feature cannot be achieved by common GPP designs, since there is just a single memory port in most architectures. Even recent designs (e.g., Intel Nehalem architecture [Inte09]) provide a maximum of only three ports that are shared by up to six dual-threaded processor cores. Still, a single core of these multi-core GPPs can use only one port at a time due to a single cache connection.

The violation of program order, which results in RAW, WAR, and WAW hazards, has to be resolved when the accesses involved become eventually non-speculative (i.e., for each surviving program branch all speculated values are known and guaranteed to be correct). With distributed memory nodes at instruction granularity, the individual accesses have to be allocated in the correct order and to the correct instruction. To this end, the commit/cancel scheme described in the previous Section 7.1.2 enables fine-grained *transactional* memory accesses [BGHS08, GCMG94, HWCC04], which can be used to enforce program order by labelling each transaction with a strictly increasing *sequence number* (shown as **seq n** in Figure 7.14). Each transaction represents an atomic operation, which is serialized internally. The memory accesses are then logically treated in program order, both before and after commit (or cancel), whereas their actual execution time can be scheduled freely. Note that the sequence numbers for **write d** and **write e** could also be reversed since both writes are mutually exclusive by original program logic (Figure 7.11).

In order to fully exploit the potential of the Comrade execution model for efficient parallel datapath execution, either to accomplish high performance or lowest power computation, an extension and enhancement to the standard GPP memory system is

required, which will be elaborated in the next section.

7.2. Memory System Requirements

This section will compare two approaches for a speculative memory system, which differ significantly in complexity. The first approach suggests to implement a subset of the complete functionality, restricted to speculating only non-destructive (reads as well as control-speculated, not-yet-committed writes) memory operations, but with moderate resource utilization. In addition, the second approach provides a solution for full-featured destructive read/write speculation at higher implementation costs.

7.2.1. Lightweight Non-destructive Speculation

This lightweight approach aims to reach two goals: providing satisfactory speculation speed-ups while using a minimum of HA resources. By avoiding a speculation buffer, the major part of HA area can be saved (see next section) while retaining most of the functionality. The non-destructive speculation comprises data, address, and control speculation for all read operations as well as those write operations, where data in architectural memory has not been overwritten yet. These writes have to be transformed into non-speculative conventional write operations before they are allowed to proceed. The memory system in this configuration can provide:

- Distributed, coherent, and independent memory ports
- Reads: per-port control, address, and data prediction
- Writes: per-port control and address prediction, write must become non-speculative before data is delivered
- Best effort parallel accesses, automatic arbitration if multiple nodes share one memory port
- Simple protocol, including indication for speculative nature of accesses
- Accesses can be completed (committed) or cancelled
- Efficient, resource-saving implementation

7.2.2. Speculating Memory Writes: Load Store Queue

While pure non-destructive speculation can be realized with relatively lightweight and efficient hardware structures, destructive memory *write* speculation requires significantly more functionality to be incorporated within the memory system. As described above, both control and data speculation require multiple versions of the original memory contents to be preserved for later restoration in case of a failed speculation. Although a

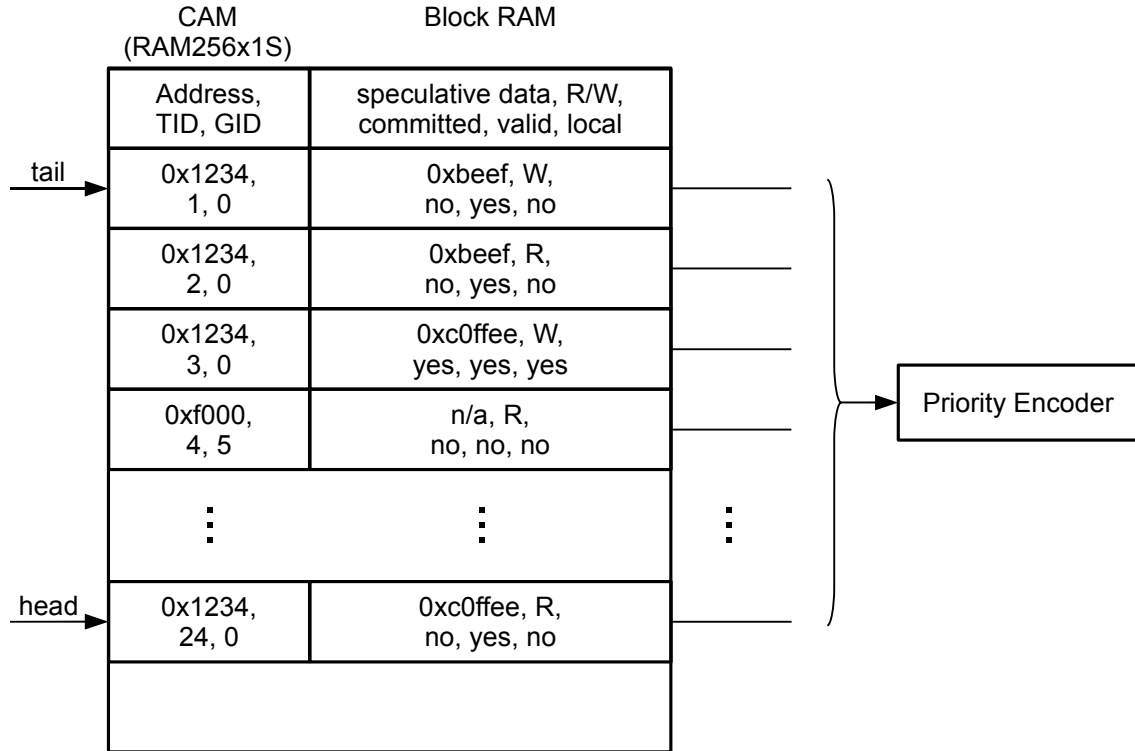


Figure 7.15.: Load Store Queue Data Structure

write buffer (described in Section 7.1.3) theoretically provides such functionality, it cannot handle reads and thus does not provide store-to-load forwarding. Furthermore, it does not scale to many memory ports, similar to most of its competitors, such as log-based transactional memory [MBMH06, YBMM07] and multi version caches [GVSS98].

These approaches suffer from one or both of two limitations: First, a centralized structure inhibits simultaneous updates to the current speculative memory operations from multiple memory nodes (e.g., log-based TM). This bottleneck is unacceptable in a highly-parallelized execution model. Second, the logic that has to be replicated for every memory port to enable simultaneous accesses, as well as the required coherence traffic volume grow too large. Such is the case with multi version caches, which include per-node Load Store Queues (LSQs). The LSQ ([SDBM03] presents an overview of the different types) offers a hybrid solution using a logically centralized structure, which is however replicated for each memory port at moderate per-port resource costs. Nevertheless, the LSQ shall serve as a representative example for the complexity of full-featured speculative memory systems, and the relatively high resource utilization compared with the lightweight approach described in the previous section.

Similar to a write buffer, an LSQ keeps a record of all speculative memory write accesses, but in addition also includes speculative reads. Its core (shown in Figure 7.15) consists of a combined Content Addressable Memory (CAM, implemented efficiently on Virtex 5/6, e.g., using RAM256X1S memory primitives and dedicated fast carry chains similar to [BrNe99]) and RAM (using, e.g., Virtex 5/6 Block RAM primitives),

7. Speculative Memory System

with subsequent priority encoders. The CAM is set up to be searched for addresses, Transaction Identifiers (TID), and Group Identifiers (GID), or any combination of these. The TID is the sequence number of an access representing the original program order. It is unique in a speculative program section. Hence, speculation has to be stopped before every TID overflow, completing all pending speculative accesses and restarting at TID zero. Note that the maximum overall speculation depth, which is limited by the TID bitwidth, is not to be confused with the number of LSQ entries. To provide for speculative execution of longer-running program parts without resetting and restarting speculation too frequently, thus diminishing speculation gains, a width of 10 bits (yielding a maximum speculation depth of 1024) was chosen for this example.

Above the TIDs, accesses in the same control branch of a program share a common GID. Since speculative writes must not be visible outside of their control branch/group, the GID is used as a discriminator for WAR/RAW resolution (see Section 7.1.2). To accommodate a whole program tree of control branches, it is advisable to provide for greater GID values, which were chosen in this example as 8 bits wide. Both TID and GID have to be provided by the user when an access is initiated via a memory port.

Whenever an address/TID/GID key combination is found at a certain position (LSQ hit), additional information (which cannot serve as a search key) is instantly available from the RAM (also cf. Block RAM in Figure 7.15):

- The *speculative data* itself
- Type of access is *read* or *write*
- Access has been *committed* (not equivalent with ready to retire: WAW)
- LSQ entry is *valid*
- Send speculation updates on this port only to accesses *local* to this port

The separation between CAM and RAM is made due to the much larger resource requirements of a CAM compared with a RAM of the same depth. Thus, only the searchable keys have to be stored in the expensive CAM. The newest or oldest transaction that belongs to a certain group can be found by selecting the whole group with the CAM and applying the respective priority encoder on the TIDs afterwards. Thus, queries can be formulated for, e.g., the latest access to a certain address, accesses that can be retired, or all accesses that are older than a given transaction.

The LSQ is organized as a ring buffer with **head** and **tail** pointer. During normal operation, every transaction is stored in the LSQ at the relative position which corresponds to its TID. Thus, holes (marked by an inactive **valid** flag, TID 4 in Figure 7.15) may exist temporarily in the queue when transactions are initiated out-of-order. There are other possibilities for the sort order (not shown here, e.g., by address [FrSo96]), but these do not alter LSQ operation in general. Transactions are retired by resetting their valid bits and, optionally, writing their data to architectural memory. LSQ buffer space is then effectively freed by advancing the tail pointer of the ring buffer over empty (= invalid) entries.

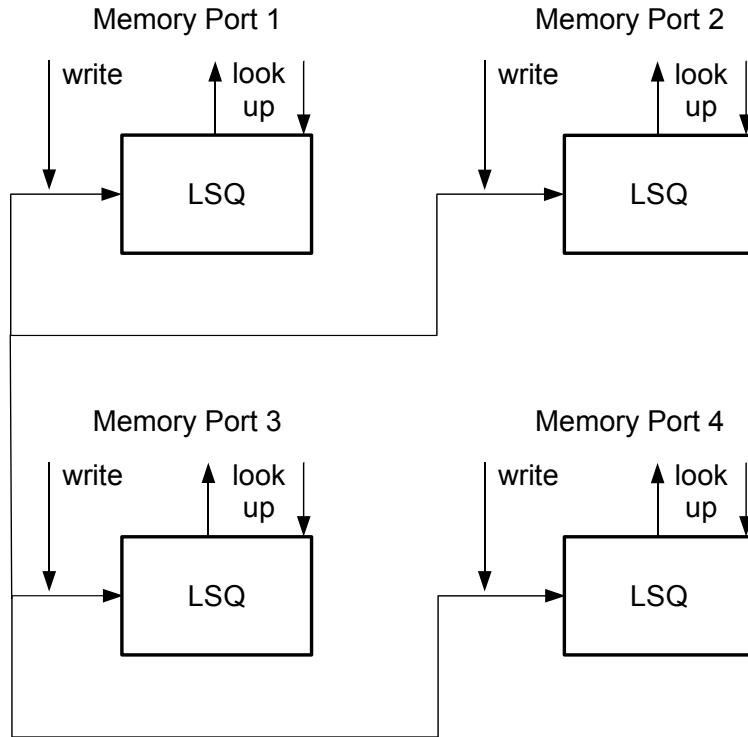


Figure 7.16.: Distributed LSQ with parallel look-ups and serialized updates (writes)

The strategies to resolve RAW/WAR/WAW hazards are similar to those of the write buffer. Read accesses must be held in flight until all older (equivalent to a lesser TID) writes to the same address have committed (TIDs 2 and 24 in Figure 7.15). If any address is speculated, all later (= greater TID) reads must be held open (not shown here). Write accesses must always be held in flight until they have committed (TID 1). In addition, committed writes (TID 3) have to be kept in the LSQ if there are older speculative writes to the same address (TID 1). Thus, correct data can be delivered to subsequent (future) read accesses (TID 24). Write accesses are not forwarded to architectural memory until a write transaction is finally retired. As a consequence, transactions are always retired from oldest to newest transaction IDs. In other words, the tail pointer of the LSQ ring buffer moves forward and retires transactions, it waits at transactions that cannot be retired yet (TID 1). Chunks of unused or invalidated transactions (e.g., groups that have been discarded because the respective control branch has not been taken) are skipped in a single step.

To support multiple distributed memory ports (as mandated, e.g., by the Comrade execution model described in Section 4), the LSQ is replicated locally for each port (a system view is shown in Figure 7.16). Thus, individual look-ups can be performed in parallel on all ports, whereas updates to the local copy need to be propagated (via the signal *write* in Figure 7.16) to keep all other copies synchronized. Despite the dual ports of the Block RAM, it is desirable that updates be performed only rarely and in the background when the LSQ is idle.

The LSQ serves as a speculation buffer by capturing all versions of speculative accesses, enabling an associated speculation controller to exert full control over the entire speculation domain (equivalent to the instruction window in the GPP execution model, see Section 7.1.3). The details of the speculation controller are beyond the scope of this work, since the LSQ data structure described above already provides a good estimate of the resources required, which is the main objective of the following discussion.

7.2.3. Discussion

As sketched in the previous section, a full-featured speculative memory system which supports reads and writes on multiple memory ports uses a large part of the available HA resources even on modern FPGAs. The largest part of an LSQ in terms of HW area comprises the CAM and its associated control logic. While the size of the controller cannot be estimated reliably without actually implementing it, the sizes for the CAM can be derived from the previous descriptions with sufficient accuracy.

Table 7.1 shows the Virtex-5VFX70T chip area requirements for LSQ CAMs (without control logic) of various sizes. The CAM implementations were generated employing Xilinx Core Generator 11.4 and subsequently mapped with ISE 11.4 to obtain the maximum clock frequencies. To this end, the CAM width was calculated from the address (32 bits), TID (10 bits, see previous section), and GID (8 bits, 256 groups for control tree hierarchy) search keys from the previous section, totalling to 50 bits per entry. Assuming that control blocks are essentially equally sized in both SW and the Comrade-HA, the number of entries per LSQ corresponds to the write buffer sizes of modern GPPs [ShLi05, Table 8.2] or to the equivalent register renaming set [HePa06, Figure 2.27]. Register renaming is a similar technique for memory disambiguation [HePa06, p. 127] which is applicable to register machines only, but not to the HA datapaths discussed here.

While the RAM area is negligible (e.g., with the RAM configuration described in the previous section, the maximum of 256 entries with 32 bits data + 4 flags = 36 bits each still fits in a single Block RAM), the CAMs need considerable area, up to 76% of 5VFX70T in a typical configuration featuring eight memory ports. Even with a moderate four memory ports at 128 LSQ entries or eight ports at 64 entries, one fifth of the available chip area is occupied for non-datapath logic. Note that the LSQ controller logic must still be added to the already unacceptable area consumption. In all configurations,

Entries	Area (LUTs)			Total 5VFX70T			Min. Period (ns)	Max. Clock (MHz)
	# Memory Ports			# Memory Ports				
	1	4	8	1	4	8		
32	586	2,344	4,688	1%	5%	10%	5.82	172
64	1,120	4,480	8,960	3%	10%	20%	6.43	155
128	2,171	8,684	17,368	5%	19%	39%	8.33	120
256	4,277	17,108	34,216	10%	38%	76%	9.40	106

Table 7.1.: LSQ CAM area and timing for typical numbers of entries and memory ports

the design fails to achieve the frequency goal of 200 MHz for the Virtex-5 rSoC even when not considering the controller logic, which would further delay the critical path. Assuming an allowance for datapath plus controller logic of at least half of the total timing budget, the attainable clock frequencies fall below the acceptable minimum of 100 MHz, which is the baseline speed of Virtex-5 vendor reference designs.

While the LSQ with its full-featured read-write control and data speculation as well as disambiguation capabilities is functionally superior to the lightweight approach described in Section 7.2.1, it has the crucial disadvantages of large resource consumption and timing budget violation (however, future FPGA generations may be able to accommodate rSoCs with speculative multi-port memory systems based on LSQs). Since, on average, there are from twice to 10x as many reads in SW programs as writes ([HePa06, Figures B.13, B.27, B.28] [StKA99], and in benchmark HAs generated by Comrade the read-to-write ratio ranges from 5:4 to 146:1 [GäSK08, Gädk10]), most of the potential for acceleration can still be leveraged by read-only speculation. In this lean system, no write buffer or LSQ is needed, as speculation remains non-destructive. Note, however, that non-destructive write speculation is provided as well at no extra cost. As will be shown in Section 8.3, non-destructive speculation achieves almost as good speculation gains as the full-featured LSQ, while at the same time being more efficient both in terms of area and performance. The next section will present the implementation details of this solution.

7.3. Efficient Implementation

As has been shown in the previous section, distributed per-port LSQs, used as a representative example for implementing a full-featured speculative memory system, are a powerful but rather costly approach. Instead, the remainder of this chapter will be concentrating on lightweight non-destructive speculation, which will be shown to be both effective and efficient in Section 8.3. As a first step to assess the possible performance gain using non-destructive speculation, a control speculation scheme for multiple memory ports (described in Section 7.3.2 below) was developed as part of this work, excluding data speculation for the time being. In the current implementation, control speculation is supported by control prediction, which employs a pattern-history predictor (described in Section 7.3.2.1). In addition, employing the architectural memory caching facilities of the underlying platform, a mechanism for cacheline prefetching was developed and implemented in order to improve speculative cache performance.

The next section gives an overview of the employed target platform and the system-level integration of the speculative memory system.

7.3.1. Target Platform

The speculative memory system is integrated as a front end to the Memory Architecture for Reconfigurable Computers (MARC, described in detail in Section 4.5.1), into the same existing and well-tried platform Xilinx ML310/ML507 (see Section 4.3) as FastLane+ (see Section 4.5), FastPath (Section 4.4), AISLE, and PHASE/V (Sections 4.6.2 and 4.6.3).

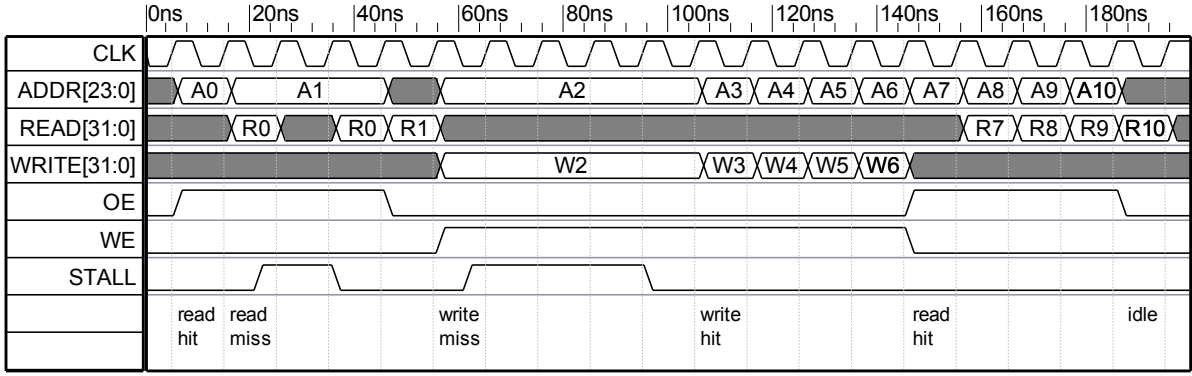


Figure 7.17.: MARC CachePort interface with alternating reads and writes

Thus, it can also leverage these effective technologies. Specifically, MARC provides a configurable, fully associative write-back caching mechanism for irregular memory access patterns, with cacheline sizes ranging from 8 words (32 bits per word) to 32 words, at a total of 32 to 128 cachelines. This cache is coherently accessible in parallel via an arbitrary number of user interfaces, called *CachePorts*. MARC provides automatic arbitration and prioritization among these ports. In the following context, the term *user* denotes a hardware circuit that is connected to MARC's CachePort interface.

The new system builds upon MARC's technology and CachePort user interface protocol [Lang01]. MARC offers a simple double handshake interface, which seamlessly integrates with automatically generated datapaths, but can also easily be attached to hand-crafted IP cores (cf. Chapter 6). Figure 7.17 shows a sequence of alternating reads and writes resulting in cache hits and misses, with minimum turn-around cycles in between. The protocol comprises a user-driven read request signal **OE** (Output Enable, shown in the figure), which must be activated until the **STALL** signal is reset by MARC. The assertion of **STALL** indicates that read data is not yet available. Simultaneously with **OE**, the user must present the read address via the **ADDR** signal. Read data is then delivered one clock cycle after the associated **OE-ADDR** pair. Likewise, write accesses set **WE** (Write Enable) and **ADDR** until **STALL** is reset. Unlike read data, which is delayed by one clock cycle, write data must be presented by the user simultaneously with the **WE-ADDR** pair.

The speculation functionality is implemented as an extension to the MARC user interface protocol. To this end, new signals with associated semantics (described in detail in Section 7.4 below) were defined, which support control speculation (see next section) with access prediction as well as cacheline prefetching (described in Section 7.3.2.2). However, MARC's classic non-speculative functionality is retained unchanged as a subset.

7.3.2. Lightweight Control Speculation

The lightweight speculative memory system, which was developed as part of this work, combines two independent systems: control speculation with access prediction (see next section), and prefetching (described in Section 7.3.2.2). The system is implemented as a

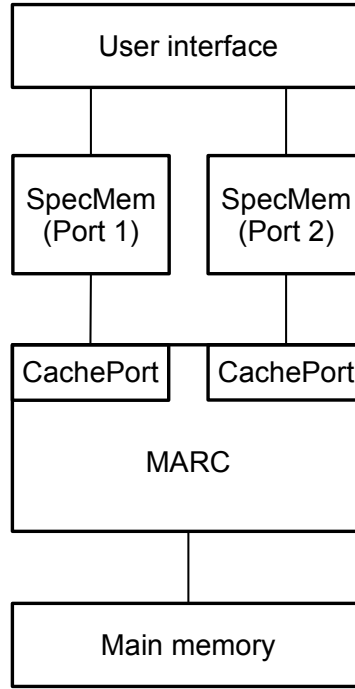


Figure 7.18.: Speculative memory ports are a front end to MARC’s CachePorts

front end to MARC, attaching to its CachePorts, thus creating a new memory hierarchy level (see Figure 7.18).

However, it also closely interacts with MARC via additional private interfaces to control some of MARC’s internal policies, e.g., for access prediction. Apart from this new private interface, MARC was modified to accept read and write access cancellation, which results in aborted cacheline transfers. Thus, failed speculative operations do not block the memory interface for longer than necessary. To this end, MARC’s internal state machines and Content Addressable Memory (CAM, required as cache tag memory) implementation were modified to provide as many termination points as possible, while maintaining their high level of concurrency for parallel memory accesses via multiple distributed CachePorts. The cancellation of speculative accesses is indicated by two new signals at the user interface, whereas transaction commits are signalled using MARC’s legacy read (OE) or write (WE) mechanisms. Both functions will be described in greater detail in Section 7.4.

The new speculative memory system is designed to provide one port per memory access in a speculative program section. Hence, a bijective mapping between memory operations and ports can be achieved to gather localized speculation statistics, e.g., the commit ratio of a memory operation (and thus the ratio of the enclosing control branch being taken) as indicated by the new control speculation signals. These per-port commit/cancel statistics are fed to the access prediction and prefetching systems, which will be explained in the next section.

7.3.2.1. Access Prediction

The mechanism for access prediction used in this implementation is tightly coupled with control speculation. In contrast to common GPP architectures, the parallel execution model enables collection of fine-grained per-port commit/cancel statistics, which represent for each memory access the probabilities for successful speculation. To this end, information about committed and cancelled memory accesses is gathered at the user interface of each memory port. The individual commit/cancel events are fed into one two-level GAg predictor per port, which consists of one or more pattern history(ies), and pattern-history table(s) containing multiple N-bit predictors. The three-letter (e.g., GAg) classification scheme for two-level predictors was introduced by Yeh and Patt [YePa92], here standing for a single **G**lobal pattern history, **A**daptive predictor, and a single **g**lobal predictor table.

In the first step, a two-level predictor constructs a pattern history, which is represented as a bit string stored in a shift register (shown in Figure 7.19 a). Each boolean decision whether a memory access was committed or cancelled is added to that history as a single bit by a simple shift operation. The pattern history is truncated by removing the oldest bit when the maximum history length is exceeded. In the second step, the pattern history is interpreted as an index into a pattern history table (PHT, Figure 7.19 b), and a single N-bit predictor is selected from the array. The N-bit predictor is an N bit wide probability value, which is the result of the two-level predictor. As an example, a 2-bit predictor can have the following states:

0: very improbable

1: improbable

2: probable

3: very probable

In the context of the speculative memory system, it represents the probability for committing a memory access. After the outcome of the predicted event is known, in this case whether an access was committed or cancelled, the N-bit predictor is updated by adding one for a commit and subtracting one for a cancel, and is subsequently written back to its place in the PHT. Overflow is prevented by limiting the maximum (3) and minimum (0) values.

Since the speculative memory system is a front end to MARC and the rest of the system, it should delay the critical path as little as possible. To this end, the GAg predictor was chosen as the simplest form of a two-level predictor with a single global history and table. Nevertheless, it yields a prediction accuracy of approximately 75% [Mart07] in the SPEC benchmark [SPEC06] with 4 bits pattern history length, 2 bits predictor width, and a single memory port for all accesses (GPP). Thus, it outperforms the primitive constant (69%) or standalone N-bit (70%) predictors [Mart07]. Two-level predictors with multiple address-dependent pattern histories or tables (SAs set-associative, PAp individual, or any other combination) use too many logic levels for single-cycle look-ups and are thus

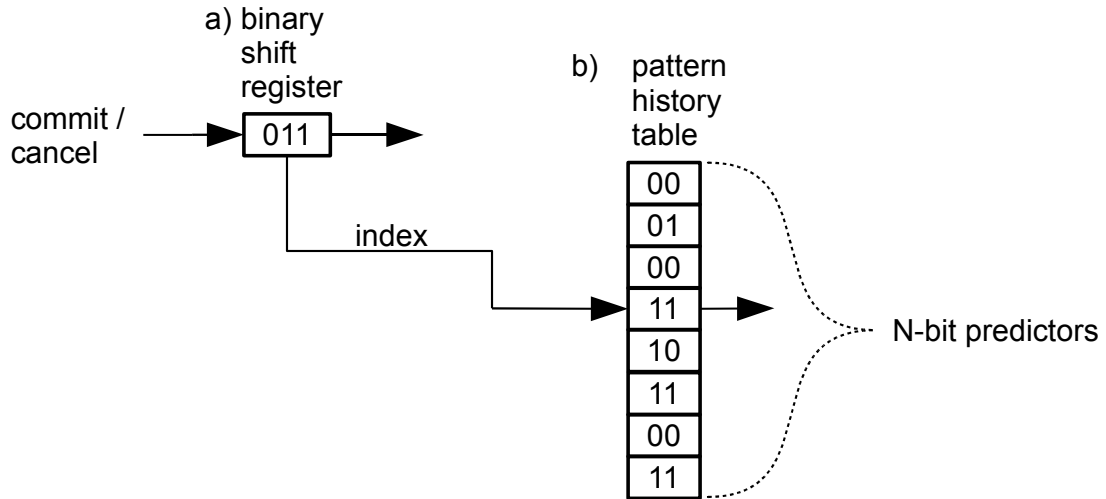


Figure 7.19.: Two-level GAg predictor

not an option. However, as the commit/cancel information is already localized due to the distributed per-access memory ports, additional address information is not required as opposed to the single-ported GPP scenario, which uses Branch Target Buffers (BTBs) to identify and predict a certain branch. Hence, the predictor accuracy is even better, which will be shown in Chapter 8.3. The 4 bit history length, 2 bit predictor GAg configuration was chosen for this implementation due to its best match characteristics at still low resource and timing budget requirements.

After the two-level GAg predictor has determined the probability for a successful memory access, the probability value is employed to set a *dynamic priority* for the corresponding port. Normally, MARC uses a fixed-priority scheme for its ports which is configured at compile/synthesis time. The new dynamic priority reflects the probability of the different speculative memory operations to succeed, i.e., to commit, by preferring those accesses with the highest commit probability. To this end, the individual predictor

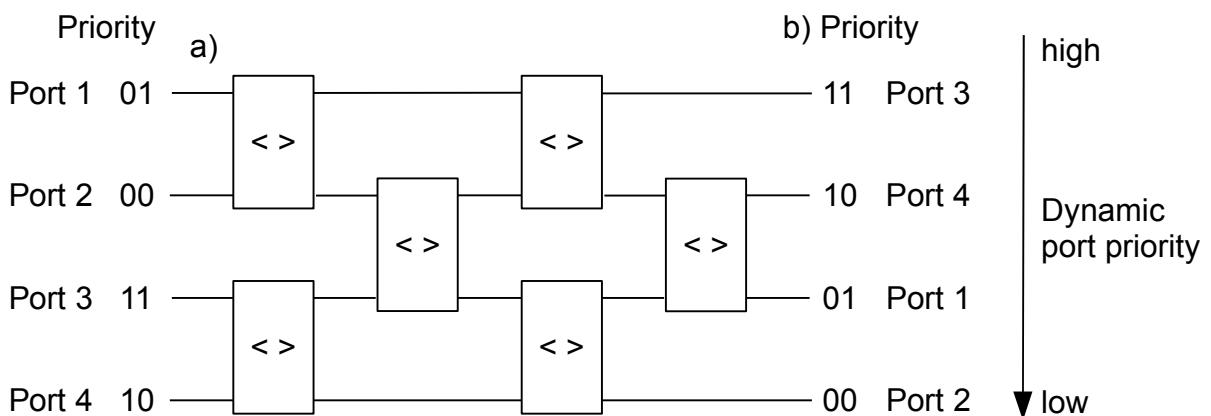


Figure 7.20.: Bitonic sorting network assigns dynamic priorities

probabilities of all distributed memory ports are arranged by a bitonic sorting network [Batc68], which is implemented as a combinational parallel HW circuit (see Figure 7.20 a). MARC's port priorities are then adjusted to dynamically reflect the associated probabilities (Figure 7.20 b). Note that the priority arrangement of the memory ports may now change dynamically from access to access. Thus, speculative accesses that are unlikely to commit have to wait until speculations with higher commit probability have completed. As the lower-probability accesses have probably been cancelled by then, no precious architectural memory bandwidth is wasted on cancelled accesses, and high-priority computations are not unnecessarily delayed. Prioritizing access prediction has also successfully been combined with the advanced MARC II [LaWK11].

7.3.2.2. Prefetching

In conjunction with access prediction described in the previous section, cacheline prefetching is another efficient method to further enhance memory bandwidth utilization. To this end, portions of the architectural memory, which cover addresses of speculative accesses with high commit probabilities, are loaded into the cache *prior* to the actual execution of the access. By thus hiding the cache latency, the data is already available in the cache for read or modification when the access is started. While prefetching is not new, it has never been applied in combination with a speculative memory system, especially when using multiple distributed memory nodes/ports in a parallel execution model.

Apart from initiating speculative accesses, MARC's new user interface signals (described in the next section) are used to control cacheline prefetching as well. Such application-initiated prefetching is known as *software prefetching* in the GPP domain [KaYe05, Chapter 7.2], which could be better termed *HA prefetching* here. Whenever a speculative access is initiated, the cacheline which corresponds to its address is prefetched subject to the priority determined by the GAg predictors described in the previous section. Hence, a cacheline is only prefetched if no other pending accesses (speculative, non-speculative, and prefetch) with higher priority are adversely affected. On the other hand, a high-priority prefetch may supersede a low-priority access, which again, by early prefetching, expedites accesses with a high commit probability over those which are unlikely to commit.

However, a prefetching cacheline transfer can also be aborted before completion, when the access which has initiated the prefetch is cancelled at the user interface. Thus, the architectural and cache memory bandwidths are sooner made available to other accesses again. MARC's internal state machines were modified to ensure that no data is lost or accidentally overwritten due to premature cacheline transfer aborts. Nevertheless, transfers may be aborted at any time, a single abort requiring two clock cycles on average. Prioritized prefetching has been successfully integrated with MARC II as well [LaWK11].

To enable access cancellation at any time, the CAM implementation of MARC's cache had to be changed from shift-register (Xilinx SRL16E primitive) to RAM (RAM16X1S, see Figure 7.21), since the SRL16E implementation requires 16 clock cycles to write or erase a cache tag [BrNe99]. In contrast, the new RAM implementation merely needs a total of two cycles to erase a tag (one for retrieval of the old tag from tag RAM, and one to erase it, signal `write=0` in Figure 7.21), and three for writing a new tag

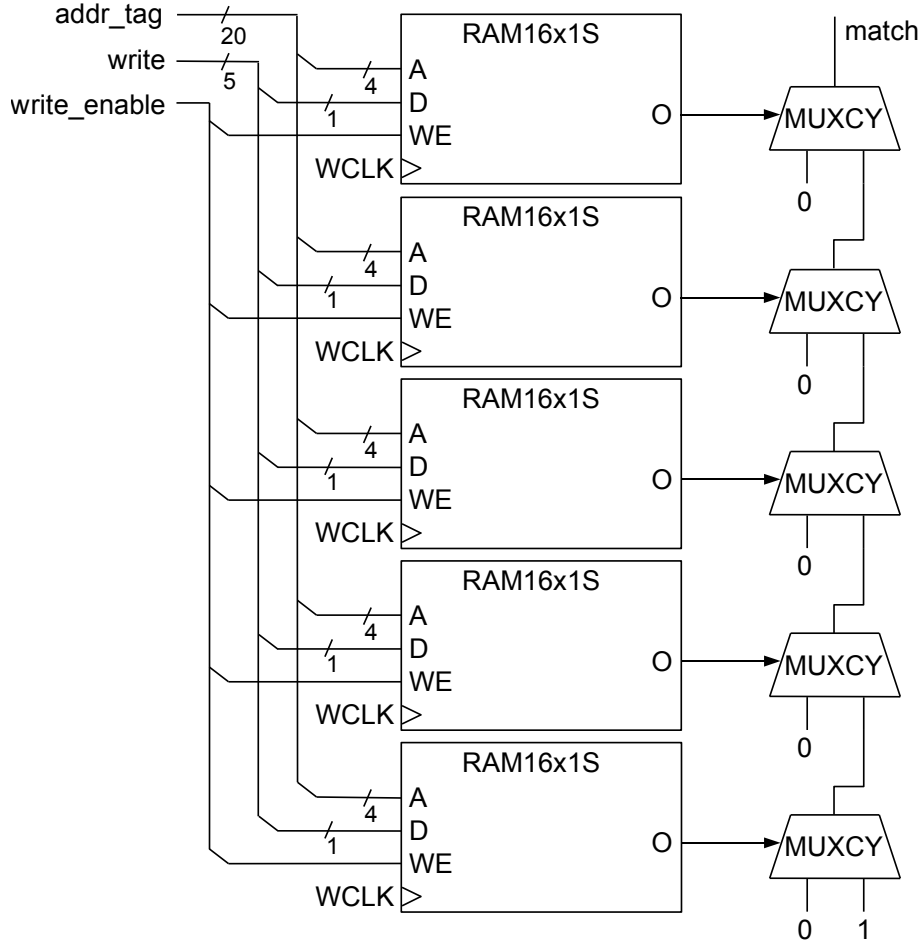


Figure 7.21.: A single CAM cell with fast write capability

(one additional write cycle, signal `write=1`). It was specifically tailored to the needs of the speculative memory system while maintaining optimal compatibility with MARC's conventional cache functionality. Single-cycle tag look-up operations (via signal `addr_tag`) are unaffected by this modification and continue to function identically: The output of each `RAM16X1S`, which stores a one-hot encoded partial tag, is logically ANDed (signal `match`) via fast carry chain multiplexers (`MUXCY`). Technically, the LUTs were switched from shift register mode to RAM mode to accomplish the change, thus leaving both area requirements and floorplan untouched.

The next section describes the integration of the speculative memory system with FastLane+ and MARC in greater detail.

7.4. System Integration: FastLane+ and MARC

As explained above, the speculative memory system is implemented as front end to the conventional MARC CachePorts. To this end, three new Verilog modules were

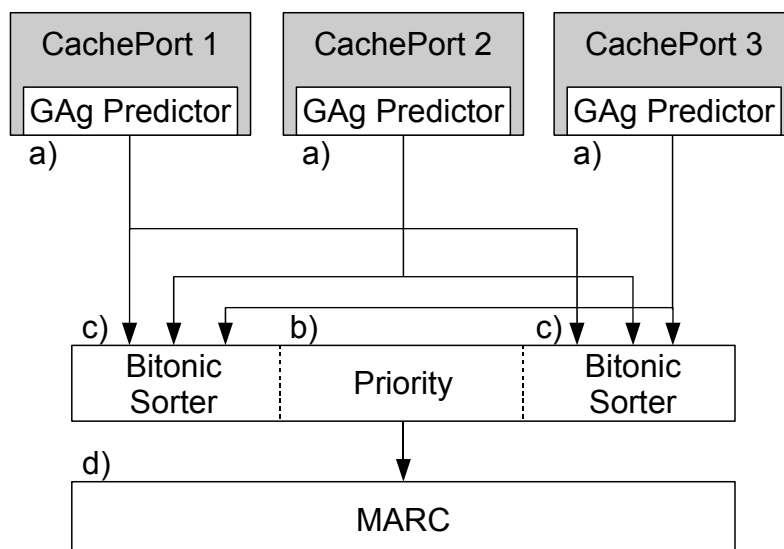


Figure 7.22.: New MARC front end for prioritizing control speculation

developed, and several existing MARC modules were modified to accommodate the new functionality.

For each memory port (CachePort), a separate *GAg predictor module* is instantiated to collect per-port commit/cancel statistics of speculative accesses (shown in Figure 7.22 a). The probability predictions for further accesses to the individual ports are processed in the central *Priority module*, which is instantiated at MARC's top level (Figure 7.22 b). Depending on the number of internal cache memory blocks (default is two), the Priority module is automatically configured before synthesis to instantiate the required number of *Bitonic sorters* (Figure 7.22 c), which are coordinated by their parent to calculate the new dynamic CachePort priorities (see Section 7.3.2.1 above). The dynamic priorities are then coupled into MARC's existing priority management (Figure 7.22 d) to override the static scheme.

The *speculation controller*, which coordinates the high-level operations such as initiation, commit, or abort of a memory access (including architectural memory) as well as the prefetching mechanism, was integrated into MARC's CachePort module as an extension to the existing state machine (see Figure 7.23). Three new states (*Delete_old_tag*, *Wait_inactive*, *Invalidate*) were added to handle the write operation to the new CAM, and memory access cancellation. The speculative FSM was equipped with additional branches back to the *Inactive* state such that accesses can now be cancelled in every state. The states reflecting speculative reads (*Output Prepare*) and writes (*Write Prepare*) were incorporated into the existing procedures for conventional reads and writes. Hence, the states *Alloc_to_cacheline* and *Alloc_to_techmod* transfer a cacheline from and to architectural memory, respectively.

The next section describes the extensions to the user interface for speculative operation.

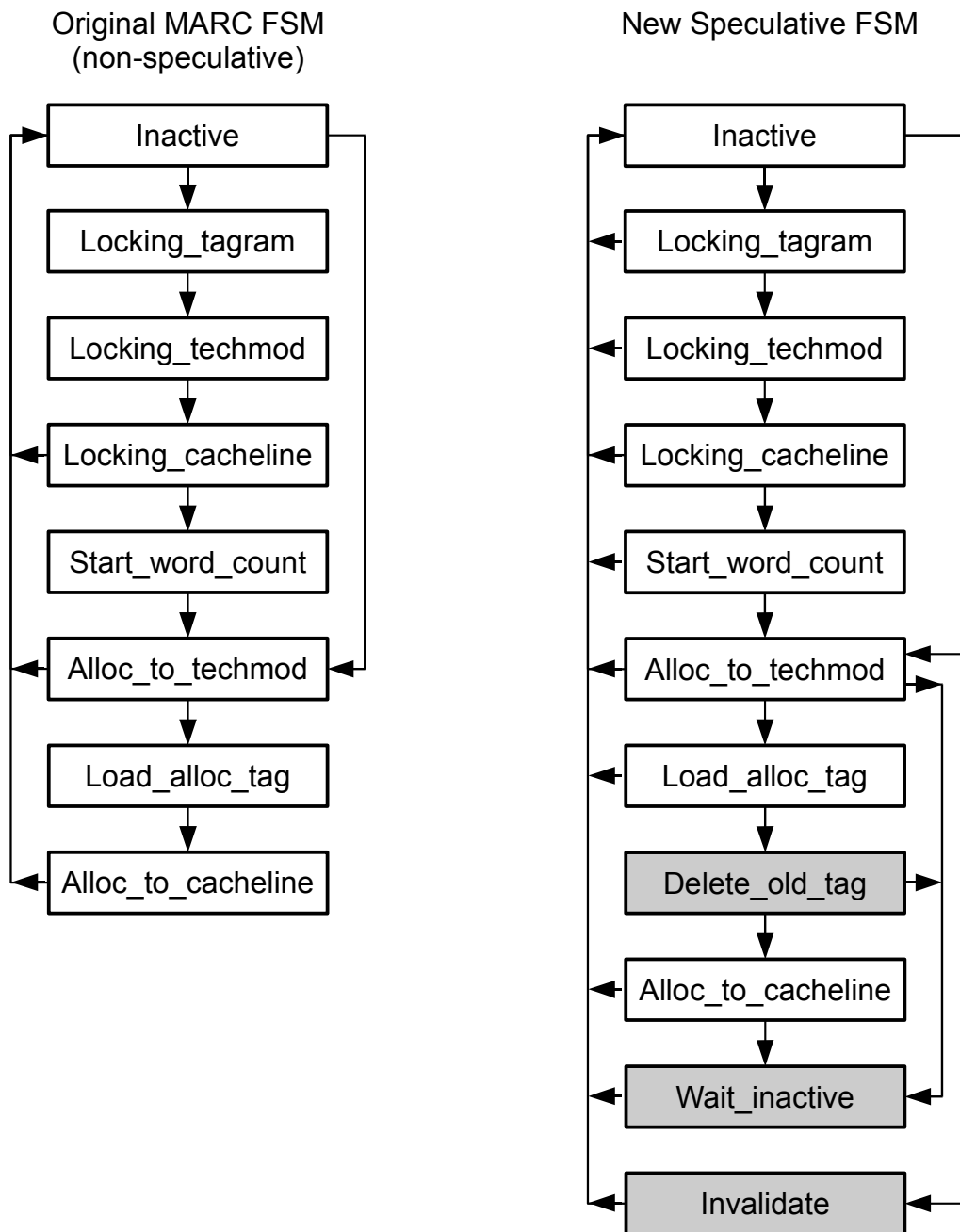


Figure 7.23.: Original non-speculative and new speculative CachePort FSM

7.4.1. User Interface Signals

As the speculative memory system was designed as a front end to MARC (see Section 7.3.1 above), the new interface extends MARC's conventional CachePort interface. Hence, MARC's semantics for non-speculative accesses (also described in Section 7.3.1) are still valid.

Control speculation allows the speculative execution of read and write accesses. The new speculative accesses adhere to a prepare-commit/cancel scheme. The prepare stage corresponds to the creation of a new speculative operation. The commit stage marks the end of speculation, address and data are now effectively valid. On the other hand, the cancel stage aborts a read or write operation as if it was never executed. In case of a read, the cancel signifies that the read data is no longer needed, the read can be aborted if it is still in progress. A write can be cancelled until data is actually written, at which point the write must commit or cancel (cf. restrictions described in Section 7.2.1).

To indicate the speculative nature of an access, two new signals were added to MARC's CachePort interface:

- **OP (Output Prepare, Input):**

Output Prepare is similar to OE: It initiates a cached read access from ADDR, but the data is not finally delivered. If OP is reset at any time, the associated read is cancelled (shown in Figure 7.24 a). On the other hand, if OE is set *additionally* when OP has already been set, the read eventually completes (indicated by STALL becoming inactive, Figure 7.24 b) and is thus committed.

- **WP (Write Prepare, Input):**

Likewise, Write Prepare initiates a write to ADDR, but does not actually write data. If WP is reset at any time, the write is cancelled (shown in Figure 7.25 a). However, if WE is set *additionally* when WP has already been set, the write eventually completes (indicated by STALL becoming inactive, Figure 7.25 b) and thus commits, writing data.

Note that there is no explicit commit signal, a commit is signalled via a normal MARC read or write access using the OE and WE signals. In addition, MARC's protocol

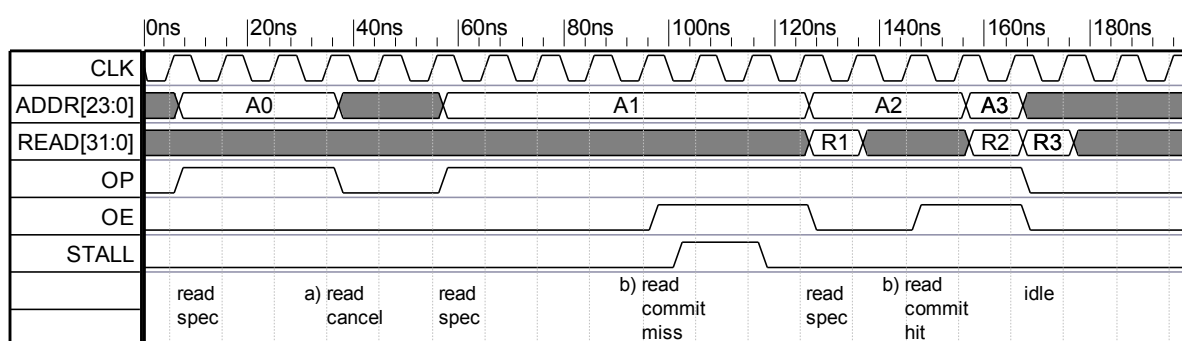


Figure 7.24.: Cancelling and committing speculative reads

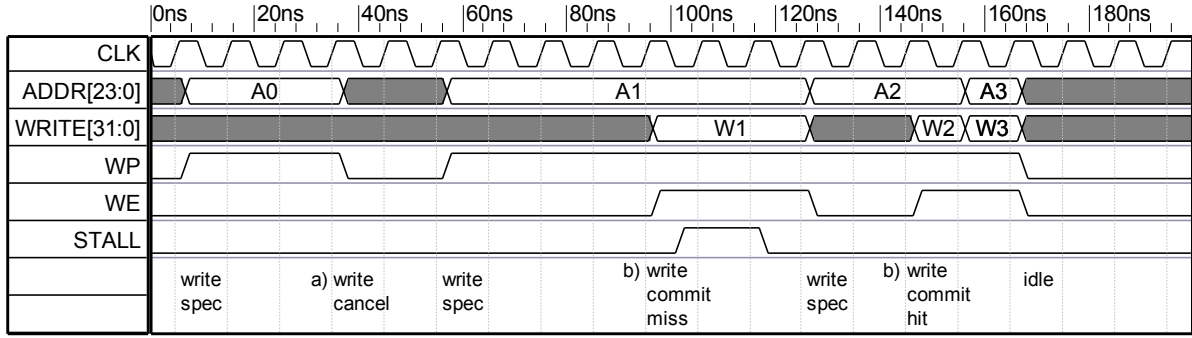


Figure 7.25.: Cancelling and committing speculative writes

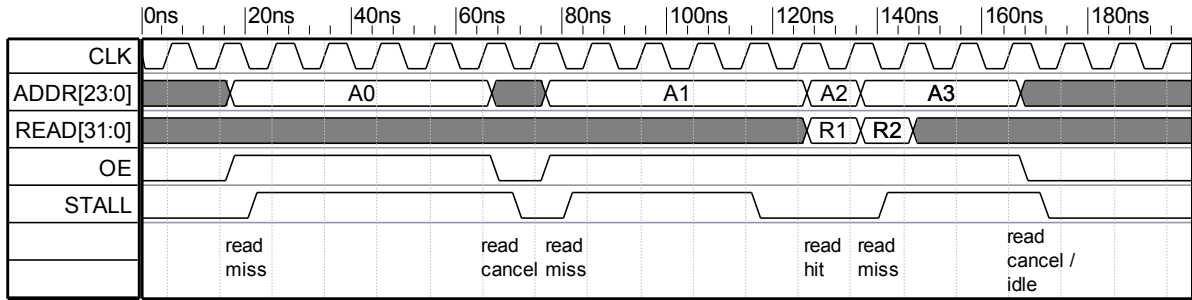


Figure 7.26.: Cancelling non-speculative reads

semantics were extended to support the cancellation of ongoing accesses by *resetting* the OE/WE signals prematurely when STALL is still set (see Figure 7.26, using the example of OE).

The next section shows how the speculative memory system benefits from statistical information conveyed by CoMAP.

7.5. System Integration: CoMAP

Although the speculative memory system was designed as a pure self-learning, auto-adaptive system, the quality of speculative predictions can profit from additional static information given by the user in advance at synthesis time. Such information can be derived from data traffic characteristics provided by CoMAP (described in Section 5). These statistics are then employed to initialize the individual access predictors (Section 7.3.2.1) of the memory system with biased values, which prioritize the different traffic flows accordingly. Apart from adjusting the predictors, CoMAP's traffic patterns can also be used to influence the new dynamic priority management (also described in Section 7.3.2.1) directly by reordering the default priority arrangement, which is applied whenever at least two predictors return the same value at a certain time. As a further step, it may be beneficial to waive dynamic priorities completely and revert to MARC's conventional static priorities, if the traffic characteristics are sufficiently significant and reliable. With certain combinations of traffic patterns (e.g., a high volume data stream among other

```

interface mixed_priorities
  type: custom
  version: 1
  port HIGH_STATIC_PRIO
    transaction: data
    direction: input
    width: 32
    sequence repeat: inf
    bigendian: 32 bit signed
  end sequence
  enableout name: ACK_STATIC_OUT offset: 0 latency: 0
  traffic linear
    blocksize: 256Ki
    burstiness: 30% burstsize: 512Ki
  end traffic
end port
  port HIGH_DYNAMIC_PRIO
    transaction: data
    direction: output
    width: 32
    sequence repeat: inf
    bigendian: 32 bit signed
  end sequence
  enablein name: ACK_HIGHDYN_IN offset: 0 latency: 0
  traffic random
    distribution gaussian: 80% range: 32Ki
  end traffic
end port
  port LOW_DYNAMIC_PRIO
    transaction: data
    direction: input
    width: 32
    sequence repeat: inf
    bigendian: 32 bit signed
  end sequence
  enableout name: ACK_LOWDYN_OUT offset: 0 latency: 0
  traffic random
    distribution local: 1Mi
  end traffic
end port
  port ...
  ...
end interface

```

Figure 7.27.: CoMAP traffic patterns inducing static and dynamic memory port priorities

memory ports with unknown properties), a mix of both dynamic and static priorities can be advantageous.

The example in Figure 7.27 demonstrates how a given CoMAP traffic pattern is transformed into a mix of two biased predictors which control dynamic port priorities, and a static high priority for a port with high traffic volume. CoMAP component definitions as well as the definitions of the handshaking ports are omitted for clarity. The port **HIGH_STATIC_PRIO** is assigned a high static priority due to its linear traffic scheme with large block sizes, which is expected to require a constant high-volume stream of data. Hence, it is exempted from MARC's new dynamic priority management and served permanently with the highest priority. Note that the **linear** traffic scheme is handled by a CachePort here, although it would typically be realized as a data streaming service (e.g., MARC's StreamPort [LaKo00]). However, CachePorts may be used if streaming services are not available or the user cannot interface to a streaming protocol. The port **HIGH_DYNAMIC_PRIO** is generated with dynamic priority management, in which the access predictor is biased towards higher commit probabilities, e.g., by initializing all 2-bit predictors in the PHT (see Section 7.3.2.1 above) to binary 11. The high commit probability is derived from the relative spatial locality of the gaussian distribution, leading to a presumably high cache hit rate. Since cache hits do not require accesses to lower levels of architectural memory, they can be granted higher priority without delaying other system activities significantly. On the other hand, the local distribution of port **LOW_DYNAMIC_PRIORITY** extends over 1 MiB of memory, which is not expected to fit into the cache. Hence, frequent accesses to architectural memory are required to flush and refill the cachelines. Since these accesses take much longer than cache hits and impose a heavy load on the memory bus(es), they are scheduled with lower priority, e.g., by initializing all PHT predictor entries to binary 00. Thus, their interference with other accesses is reduced to a minimum, while the additional delay incurred by low priority does not increase significantly their already long execution times.

7.6. Chapter Summary

This chapter presented a lightweight speculative memory system and its implementation as a vital ingredient of a high-performance, efficient Reconfigurable Computing Platform that this thesis intends to define. To this end, the main aspects of speculative program execution were identified. Control, data, and address speculation as well as prediction were then examined for their differences in suitability for single and multi-core architectures, the latter with special regard to parallel execution models for hardware accelerators. In addition, for parallel hardware accelerator execution it proved advantageous to move the speculation mechanism to the memory system. It was found that the potential gains of write data speculation come at hardware resource costs prohibitively high for current devices, but might become feasible for future generations of reconfigurable technology. Further investigation revealed that control prediction, or in the case of a memory system, access prediction is the most promising candidate for high speculation gains at low hardware cost. Thus, the speculative memory system significantly increases

7. *Speculative Memory System*

the exploitation efficiency of the high memory bandwidth provided by FastLane+. Finally, an efficient implementation was shown which saves on resources by reusing and interacting with the existing MARC system, and which integrates well with the other parts of this work that constitute the Reconfigurable Computing Platform.

8. Experimental Results

Each Column of the Reconfigurable Computing Platform is now evaluated separately to better understand its respective impact on the overall system design, operation, and performance. Platform management and the HW/SW interface are described together in a single section due to their strong dependencies. Power and energy consumption, both being important factors in the embedded system environment, are then considered for the hardware implementations of the execution model and the speculative memory system.

8.1. Execution Model

The execution model is evaluated by evaluating the impact of its different components on system-level performance. First, the advantages of the FastPath quick signalling solution over standard OS IRQ handling are shown. Then, results for the FastLane+ enhanced memory system are given. Finally, the different solutions to integrate the SW/HA communication with the OS are compared.

8.1.1. FastPath

The main contribution of FastPath is a reduced HA/SW signaling latency. Since a common aim of using an ACS in the first place (instead of a conventional GPP-only computer) is improved performance, the impact of signaling latency (communication overhead $t_{overhead}$) on the effectively achievable speed-up will be examined.

To this end, the *raw GPP-to-HA acceleration factor* is computed as $HW_{accel} = t_{SW}/t_{HA}$, the ratio of times for running an algorithm (just part of a program) in SW to that of performing the same computation on an HA. Note that this raw factor does not consider the communication overhead yet. However, the *effective speed-up factor* for that part of the program also includes the overhead:

$$\text{effective speed-up} = \frac{t_{SW}}{t_{HA} + t_{overhead}} = \frac{t_{HA} \cdot HW_{accel}}{t_{HA} + t_{overhead}}$$

The *communication overhead* is computed as $t_{overhead} = t_{IRQ} + t_{sem}$. Here, t_{IRQ} is defined as the time interval between the HA initiating an interrupt and the reaction in the user space interrupt handler (e.g., determining the interrupt cause by reading from an HA register). t_{sem} is the time between the handler setting the semaphore (described in Section 4.4) and the resumption of the main program thread.

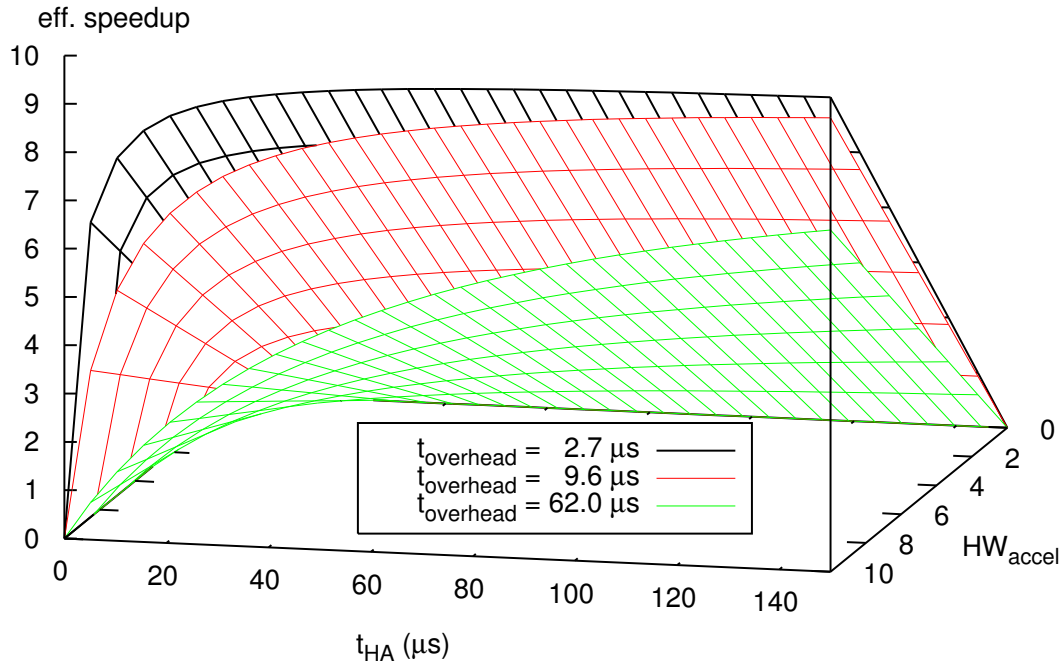


Figure 8.1.: Effective speed-up as function of HA execution time and raw HW acceleration factor for different latencies

Measured times for the different scenarios are cycle-accurate and were determined using a dedicated hardware counter. As baseline, the overhead for the standard Linux interrupt path on the ML310 (employing the usual tasklet-driven wait queue in lieu of the FastPath fast semaphore) is $t_{overhead} = 62 \mu s$. In contrast, FastPath achieves a $t_{overhead}$ between just $2.7 \mu s$ and $9.6 \mu s$ (best case: $t_{IRQ} = 2.1 \mu s$, $t_{sem} = 0.6 \mu s$; worst case: $t_{IRQ} = 8.8 \mu s$, $t_{sem} = 0.8 \mu s$). Combined with the negligible times for the live variable transfer (20...40 ns, see Section 4.4), the total overhead can thus be reduced by a factor of 6.5x to 23x (achieving low-latency GPP \leftrightarrow HA communication).

If a serialized single-threaded execution model is not required, the IRQ-handling callback function (which has full access to virtually addressed user space data and system libraries) can directly perform latency-critical operations without synchronizing to the main thread, thus creating a dedicated callback thread (cf. Section 4.4), which would improve $t_{overhead}$ by another 9–22% even over the fast register-based semaphore. In this manner, FastPath running on a relatively slow 300 MHz PowerPC 405 embedded GPP even outperforms specialized real-time variants of Linux, such as RTAI/LXRT or tuned versions of recent 2.6 kernels running on a multi-GHz desktop PC GPP [Laur04].

The system-level effects of these measurements are visualized in Figure 8.1. It shows the achievable effective speed-up (z-axis) for different combinations of raw hardware speed-up (HW_{accel} , y-axis), execution time spent in hardware (t_{HA} , x-axis), and the different communication overheads $t_{overhead}$. For the latter, the bottom surface shows the effective speed-up achievable using the standard Linux communications mechanism, while the upper two surfaces show the impact of FastPath (top: best case, middle: worst

case).

It is obvious, that the impact of an HA on total performance increases when the HA is used for longer periods of time t_{HA} and it can actually leverage the raw speed-up. Shorter values for t_{HA} , either due to smaller algorithms being offloaded to the HA, or due to the HA requesting SW Services, reduce the effective speed-up. For very short hardware execution times, the communication overhead dominates and even high hardware acceleration factors yield only small effective speed-ups or even slow-downs (effective speed-up < 1).

All three surfaces in the figure converge against an imaginary plane representing the theoretically optimal speed-up (where effective speed-up = HW_{accel}). More interesting is the behavior for intermediate values of t_{HA} . The two FastPath surfaces already reach 90% of the optimum after just $23 \dots 75 \mu s$ of hardware execution, while the conventional communication mechanism requires much longer times spent in hardware to achieve similar effective speed-ups. Such information is crucial to perform a high-quality hardware-software partitioning, since it will directly influence the granularity of the execution sections assigned to hardware or software processing. With FastPath, HW/SW partitioning can be performed at much finer granularity than using the conventional scheme: Even smaller kernels that would not yield actual speed-ups (due to the communication overhead) can now effectively be accelerated in HW.

8.1.2. FastLane+

To demonstrate the effectiveness of the *FastLane+* approach, several system load scenarios were exercised. The basic setup is identical in all cases: The actions of an actual HA are mimicked by a hardware block that repeatedly copies a 2 MiB buffer from one memory location to another as quickly as possible, totaling to 4 MiB of reads and writes per turn, and the transfer rate in MiB/s is measured. The area statistics of the vendor Virtex-II Pro (V2P) reference and FastLane+ implementations of the rSoC containing this Copy-HA are given in Table 8.1. In addition to the Copy-HA, a suite of different software programs (independent of the Copy-HA), chosen for their specific load characteristics (described below), is run on the GPP. Then, the HW execution time (the time it takes to copy memory data at full speed) and the SW execution time (the time it takes for a given program to execute on the GPP) are measured for both the original vendor-provided as well as the FastLane+ memory interface. The extreme cases of either the HA or the GPP being idle are also considered.

The suite of software programs was chosen to represent an everyday mix of typical applications that also perform I/O and calculations in main memory (instead of just running entirely within the GPP caches). The `scp` program from the OpenSSH suite [FPRS09] was instructed to copy a local file, sized 4 MiB, via network to a remote system. The same is done without encryption by `netcat` [Giac04]. The GNU `gcc` [Stal02] C compiler was evaluated while compiling the `netcat` sources.

To also cover the embedded system domain where SoC platforms similar to Virtex-II Pro are often employed, the ETSI GSM enhanced full rate speech codec [Euro00] and an image processing pipeline as often found in printers were included ([FiFY05], JPEG RGB

8. Experimental Results

HA Type	Slices	4-LUTs	Flip-Flops	Block RAM
Copy-V2P-reference	8,408	9,868	7,596	28
Copy-FastLane+	6,952	7,904	6,408	81
List-DMA	6,660	7,450	6,001	24
List-AISLE	6,664	7,451	6,009	24
List-PHASE/V	6,898	7,731	6,146	24

Table 8.1.: FPGA areas for rSoCs with specified HA at 100 MHz clock

GPP SW Load	V2P reference design		FastLane+	
	Exec time [ms]	Mem rate [MiB/s]	Exec time [ms]	Mem rate [MiB/s]
idle sys	18.81	213	5.62	1,424
scp	55.11	73	12.61	634
netcat	53.07	75	19.51	410
gcc	32.14	124	17.98	445
GSM	19.05	210	6.03	1,326
imgpipe	44.67	90	19.37	413

Table 8.2.: Copy-HA runtimes and available throughput using V2P reference design and FastLane+ memory subsystem implementations under various GPP SW loads

to CMYK conversion as part of the HP Labs VLIW Example Development Kit), both representing typical embedded applications. The various programs can be characterized as follows:

- **scp** provides a mix of GPP- and I/O load
- **netcat** exercises network I/O exclusively
- **gcc** interleaves short I/O- and long calculation phases
- **GSM** provides codec stream data processing
- **imgpipe** implements multi-stage image processing

The first set of measurements displayed in Table 8.2 considers the memory throughput of the Copy-HA under different GPP load scenarios: The time for a single 2 MiB block copy (four mega-transfers) and the resulting memory throughput are shown, both when using the original vendor-provided PLB interface as well as the FastLane+ attachment for the Copy-HA. It is obvious that FastLane+ significantly increases the HA memory throughput in *all* load scenarios, in some cases by a factor of up to 8.6. This considerably improves the high-throughput HA memory access requirement of the Comrade execution model (see Section 4.1).

The set of measurements shown in Table 8.3 quantifies the influence of running the HA at full speed on the execution times of the SW applications on the GPP. Three

GPP SW Load	HA inactive [ms]	V2P reference design [ms]	Slow-down	FastLane+ [ms]	Slow-down
scp	4,831	61,052	1,263%	5,788	120%
netcat	3,130	55,938	1,787%	3,843	122%
gcc	40,686	166,655	409%	52,526	129%
GSM	25,981	40,045	154%	27,357	105%
imgpipe	3,545	5,109	144%	3,806	107%

Table 8.3.: Software run times and slow-down on idle system and using Copy-HA attached by V2P reference design and FastLane+ memory subsystem implementations

run-times are given for each application, corresponding to the Copy-HA being inactive (not performing transfers), and the two ways of attaching the Copy-HA to the system. The column ‘slow-down’ shows the increase in SW execution time when the Copy-HA is active (e.g., a value of 107% here indicates that the application is 7% slower than with an inactive Copy-HA). The results show that, despite its high throughput to the Copy-HA, FastLane+ does not significantly impair the GPP (and thus obeys OS scheduler decisions): SW execution times are only mildly affected by the Copy-HA memory transfer, owing to the absolute priority of the GPP (cf. Section 4.5) over the HA. In contrast, the original vendor-provided reference design exhibits a steep SW performance decline, increasing execution times by a factor of up to 17.9x over that of SW running with the FastLane+-attached HA. FastLane+ thus enables the HA to access memory bandwidth that appears to be completely unused by the original memory interface.

The differences in slow-down for FastLane+ are due to the different load characteristics: **gcc** is slowed most, since it does only little I/O but spends much time transforming in-memory data structures (preempting the Copy-HA). **scp** and **netcat** perform more I/O and fewer memory accesses (less interference with Copy-HA), and are thus slowed less. **gsm** and **imgpipe** run mostly out of the GPP caches and execute almost without interference from the Copy-HA. Note that the reference design has a different slow-down profile: The HA and GPP share not just main memory, but also the PLB which arbitrates between accesses to memory and I/O devices. Thus, the more I/O intensive applications in the reference design are slowed down further by the Copy-HA than with FastLane+.

One might assume that the FastLane+ approach of giving the GPP override priority for bus access will cancel out the theoretical performance gains of FastLane+ over the original PLB-based HA attachment. However, we measured that even under these conditions, FastLane+ is able to provide the HA with roughly half of the theoretically available memory bandwidth (which is 1600 MiB/s in double-data rate mode): Practically achievable are 64b data words at a rate of 712 MiB/s, and 128b words at 1424 MiB/s, yielding a bus efficiency of 89%. At the same time, the multi-tasking OS and the SW application continue to run at almost full speed.

In scenarios where fast SW interrupt response is not required, for example, and it is possible to freeze the processor entirely (e.g. by stopping the clock signal), FastLane+ makes the full physical memory bandwidth available to the HA. This is not achievable

using the original PLB attachment, which even with a frozen processor is only able to exploit just 25% of the theoretically available read bandwidth and 33% when writing due to the limited PLB burst length and frequent re arbitration.

8.1.3. AISLE versus PHASE/V

To demonstrate the direct interchangeability of data between HA and SW (shared GPP↔HA address space) while maintaining the advantages of the FastLane+ memory system, a pointer chasing application (C code shown in Figure 8.2) was evaluated separately on both HA and SW: The application traverses a different number of elements of a randomly linked list, with each list element consisting of a 32b integer value and a pointer to the next element. On each element, the following operation is performed: If the value is odd, it is increased by one, otherwise it is kept as is. Note that such a trivial operation was chosen on purpose: This test is specifically intended to exercise the capability of the List-HA used by three of the implementations below to *traverse* irregular pointer-based data structures, instead of relying on the data streaming so common to other ACS applications. Thus, the test setup actively avoids accelerating *computation* (which would bias the results towards the HA).

Table 8.4 shows the execution times of four implementations for this application. The area statistics for the variants are shown in Table 8.1. First, a pure SW implementation was evaluated, followed by a List-HA-“accelerated” implementation using just a DMA

```

...
struct numbers {
    int number;
    struct numbers *next;
};

struct numbers *head;
struct numbers *run;
...

/* Traverse the list */
run = head;
while (run) {
    /* If number is odd, increase by one */
    if (run->number % 2)
        run->number++;
    run = run->next;
}
...

```

Figure 8.2.: C code of the pointer chasing application

List length	Software	List-HA using		
		DMA/copy	AISLE	PHASE/V
16K	7.4 ms	27.6 ms	3.4 ms	3.5 ms
32K	15.8 ms	55.1 ms	6.8 ms	12.2 ms
64K	33.0 ms	110.1 ms	13.7 ms	29.5 ms
128K	68.3 ms	220.1 ms	27.4 ms	64.0 ms

Table 8.4.: Runtimes of the pointer chasing application

Buffer, which required copying the list from the normal SW-allocated stack memory into the Buffer. A correct alignment was guaranteed between the SW-allocated memory and the Buffer, otherwise the code would also have to relocate every single pointer within the list (which can be avoided in this manner), and would be even slower. Third, the combination of DMA Buffer with the AISLE program layout was evaluated using the List-HA. Finally, employing the same accelerator again, the full virtual memory functionality provided by PHASE/V was tested.

The results show that the 100 MHz List-HA using AISLE outperforms the 300 MHz PowerPC 405 GPP by a factor of roughly 2.5, and is even 8 times faster than the conventional approach of explicitly copying data to List-HA-accessible memory (shown as DMA/copy in Table 8.4). Note again that the focus is *solely* on evaluating data access times, the potential for accelerating a more complex *algorithm* by the List-HA is deliberately not considered here.

Both AISLE and PHASE/V allow the free interchange of userspace address pointers between the HA and GPP, which allows much faster operation than the explicit copying of data between GPP and HA memories. However, the full virtual memory capabilities of PHASE/V (as all MMU-based VM architectures) do come at a performance cost. The results (Table 8.4) show that the performance of PHASE/V is highly dependent on the size of the application's data set. In the 16K case, the page mappings for the memory area completely fit into the HA-TLB, thus no thrashing occurs and the performance roughly equals that of AISLE. With 32K list elements, the performance drops since TLB thrashing begins to occur. Still, PHASE/V is 23% faster than the SW version. At 128K, heavy TLB thrashing slows the acceleration. However, even in this extreme case, the List-HA using PHASE/V is still able to outperform the pure SW implementation by 6%. Note that all pages were present in memory lest the measurements would be influenced by swap file (hard disk) performance.

Depending on the application requirements, the designer (or compiler) can thus trade-off between the simpler, more efficient AISLE and the full VM participation of PHASE/V on a per-application basis.

8.2. CoMAP and PaCIFIC

To evaluate the feasibility and efficiency of both CoMAP and PaCIFIC, their interoperability as a combined approach will be demonstrated in this section. To this end, the real

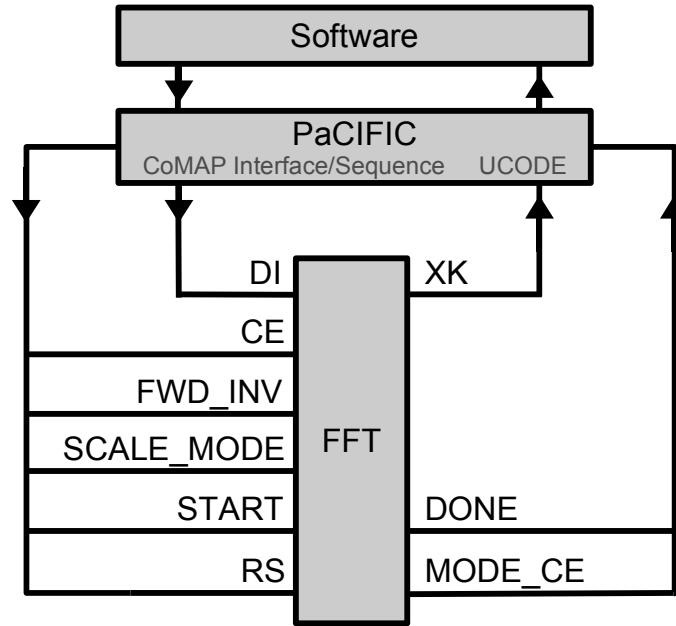


Figure 8.3.: FFT pipeline controlled by PaCIFIC (duplicate of Figure 6.13 reproduced here for convenience)

world FFT example described in Section 6.5.3 is revisited. In this scenario, the Xilinx High-Performance 16-Point Complex FFT/IFFT [Xili01] of the Core Generator suite was coupled to an ANSI C program (shown in Figure 8.3). The FFT expects data to be continuously streamed to its input buses as well as from its outputs. The output data is available after an initial latency of 82 cycles. Since the compiler does not yet include automatic support for CoMAP and PaCIFIC, the example was implemented manually using PaCIFIC techniques.

The test platform was an ACE-V ACS [Koch00] (see Figure 2.3). The relevant platform HW used here includes a 100 MHz microSPARC-IIep GPP with 64 MiB of DRAM and a Xilinx Virtex 1000 -4 FPGA. The GPP accesses the FPGA via a PCI bus and a PLX PCI9080 local bus bridge. For comparison, the FFT was also exercised on a second test platform, an Alpha Data ADM-XRC card attached via PCI to a standard PC (AMD Duron 800 MHz, 256 MiB SDRAM). The ADM-XRC is a subset of the ACE-V providing the same Virtex FPGA and PLX local bus bridge, but lacking the microSPARC processor and DRAM.

The C program executed by the processor reads the source data from a file into the DRAM, calls the FFT hardware accelerator which is implemented on the FPGA, and finally writes the result back to disk (described in closer detail in Section 6.5.3). To this end, a CoMAP description for the FFT IP core provides all static component properties such as data width, static behavior, and interface connections. The FFT datapath (shown as signals DI and XK in Figures 8.3 and 8.4) concatenates both 16-bit real and imaginary parts of the complex data values into a single 32-bit word. Data for the FFT logic is efficiently sourced and sunk by two stream engines co-located on the FPGA with a FIFO

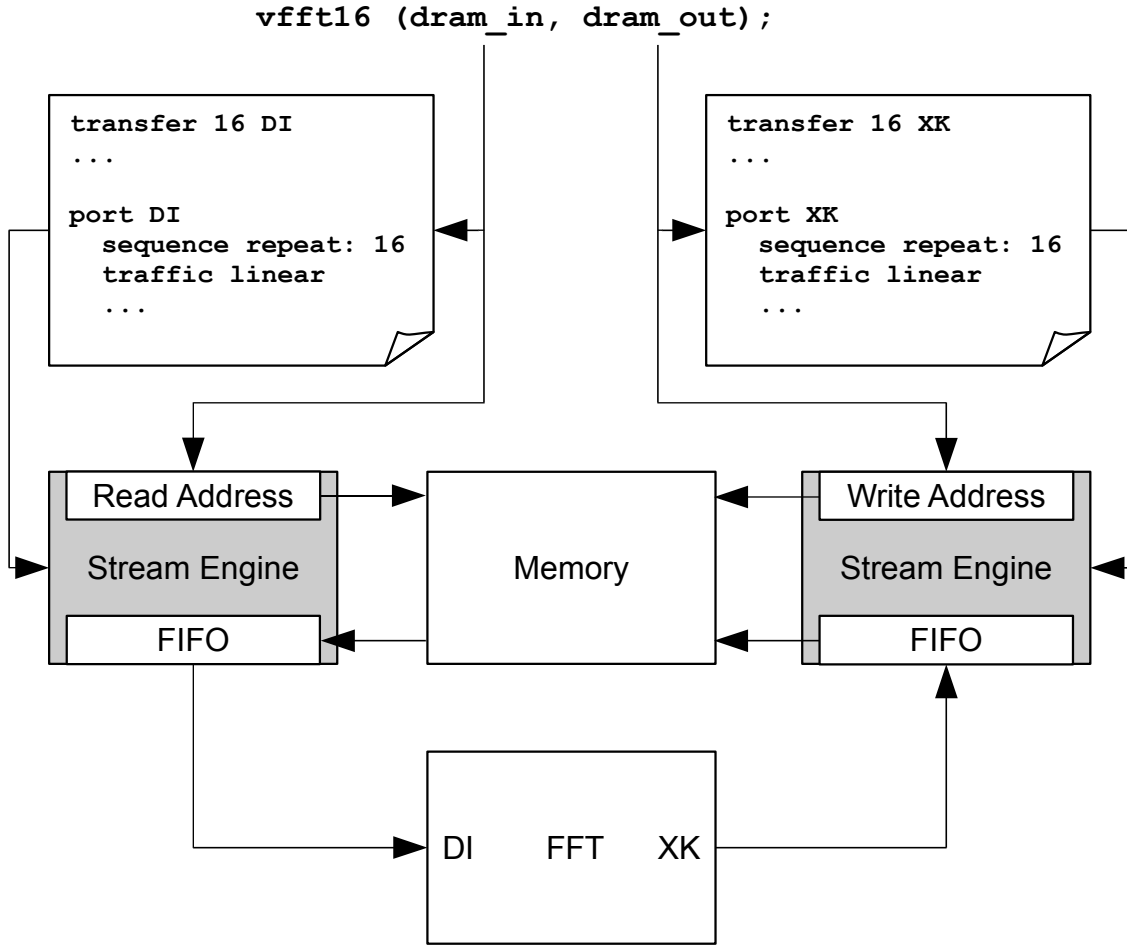


Figure 8.4.: Stream engines operating from pointers source and sink memory areas (duplicate of Figure 6.19 reproduced here for convenience)

capacity of 256x32 bit each, which access the DRAM memory in bus master mode (shown in Figure 8.4).

A UCODE block (outlined in Figure 8.3, elaborated in Section 6.5.3) for wrapping the FFT IP core controls the HW interface protocol for all core signals. After programming the operating mode, the core accepts a 16-sample block of time-domain data via signal DI. After the end of the computation is indicated, 16 frequency-domain samples can be unloaded from the core via signal XK. In a pipelined fashion, the next set of time-domain data can be provided to the core when it becomes available again.

After application of the PaCIFIC algorithms (described in Section 6.5.3), the SW part was compiled using gcc [Stal02], while the resulting RTL description for the stream engines and interface control logic was synthesized with Synplicity Synplify. It was subsequently mapped with Xilinx ISE, embedding the FFT core netlist. The achievable clock speed without optimized floorplanning for the mapping results in Table 8.5 is 30 MHz. Table 8.6 shows the performance results for the FFT application processing 4K words (32 bits each) on both ACE-V and ADM-XRC/PC at 27 MHz FPGA clock.

8. Experimental Results

	Area		Resources	
	[slices]	[total V1000]	[BlockSelectRAM]	[total V1000]
FFT	1,386	11%	0	0%
S/I*	1,385	11%	4	13%
Sum	2,771	22%	4	13%

*S/I: Stream engine and Interface control

Table 8.5.: FPGA areas for the FFT application with CoMAP/PaCIFIC

	ACE-V	ADM-XRC/PC
	[clock cycles]	[clock cycles]
S/I* read startup latency	8	8
PCI read startup latency	39	29
FFT processing	4,178	4,178
Memory transfer overhead	4,096	4,096
PCI processing overhead	8,036	6,333
PCI write flush overhead	199	58
Sum	16,556	14,702

*S/I: Stream engine and Interface control

Table 8.6.: Runtimes of the FFT application with CoMAP/PaCIFIC

The results indicate that the CoMAP/PaCIFIC combination approaches optimal FFT processing performance at moderate area overhead. To measure the performance penalty of the CoMAP/PaCIFIC interface, the execution time ratio (*effective slow-down*) between the FFT-PaCIFIC interface combination and raw FFT processing (without the interface) will be computed. The raw FFT processing time *without* CoMAP/PaCIFIC is

$$t_{raw} = t_{FFT} + t_{memory} = 4178 + 4096 = 8274 \text{ clock cycles}$$

with t_{FFT} being the pure FFT computation time, and t_{memory} being the memory transfer time. The latter reflects the serialization of read and write operations with intermediate data buffering within the stream engines during processing, mandated by the single channel to main memory on both ACE-V and ADM-XRC platforms.

The processing time *including* the CoMAP/PaCIFIC interface overhead given by $t_{stream/interface}$ amounts to

$$t_{CoMAP/PaCIFIC} = t_{raw} + t_{stream/interface} = 8274 + 8 = 8282 \text{ clock cycles}$$

resulting in an execution time overhead for the CoMAP/PaCIFIC interface of only

$$\text{effective slow-down} = \frac{t_{CoMAP/PaCIFIC}}{t_{raw}} = \frac{8282 \text{ cycles}}{8274 \text{ cycles}} \approx 1.00097x$$

The remainder of the total execution time consists of PCI overhead and latency, which differs slightly between both platforms (shown in Table 8.6). It was deliberately excluded

from *effective slow-down* calculation to not bias the result in favor of CoMAP/PaCIFIC by platform deficiencies. Furthermore, the time spent in SW processing is not considered here since it depends mostly on the host's file I/O capabilities rather than PaCIFIC interface design.

8.3. Speculative Memory System

To evaluate the effects of the different memory speculation mechanisms, four micro-benchmarks were used to examine the interaction of specific features. To this end, the manually-designed benchmark hardware was coupled to the speculative memory system (described in Chapter 7) and MARC (Section 4.5.1), the latter using a fully-associative, pseudo-random replacement cache configuration of 32 cachelines with 32 words each.

8.3.1. Linked List

First, the impact of speculation on the *Linked List* micro-benchmark (C code shown in Figure 8.5) is examined, which is an extended variant of the pointer chasing application used to evaluate the execution model (see Section 8.1.3). Tables 8.7–8.10 show the result of processing a linked list of 128K elements, 16 to 128 bytes (four to 32 words) each, using two read ports (*Rd next*, *Rd tag*) and a combined read/write port (*RW data*). The list is laid out in memory either randomly (Tables 8.7–8.9) or sequentially (Table 8.10). The processing is varied from predominantly reading (Table 8.9) to also writing in most iterations (Table 8.8), with a mix in between (Tables 8.7 and 8.10). The per-element processing time for evaluating the control condition (*Processing*) is assumed to require from two to 16 clock cycles (e.g., for a string comparison).

Due to the random nature, only limited speed-ups are achievable with speculation enabled. Moreover, MARC can serve up to four memory accesses in parallel, while the Linked List application never issues more than three simultaneous requests. Hence, no gains can be expected from pure access speculation, since all cache hits are served immediately and in parallel. This is reflected by identical execution times without speculation and with access speculation only, shown as a combined column (*Non-Spec*). However, whenever data is not yet in the cache, combined access prediction and speculative greedy prefetching [LuMo96] (*Spec+Pref*) attain net accelerations of up to 1.22x (*Accel*).

As a central observation, speculation cannot outperform the cache. This does not come as a surprise, since on a cache hit, the cache operates at the theoretical optimum, delivering data within a single clock cycle. While increasing randomness (= cache misses) and writes (= modified cachelines) increase the run time, the speculation performance improves due to the degraded cache performance. Here, speculation achieves maximum gains as it reorders cacheline transfers, or discards unneeded accesses and transfers at all. With the sequential list layout, however, a single cacheline holds multiple temporally adjacent elements (similar to streaming), which increases the hit rate. Hence, there are fewer occasions for profitable speculation. The cache performance also shrinks with increasing element sizes, when the cache can only hold small parts of the linked list, and a

8. Experimental Results

```

...
struct list {
    struct list *next;
    char data[8];
    int tag;
};

struct list *head;
struct list *run;
char *reference_data;
...

/* Traverse the list */
run = head;
while (run) {
    /* If data > reference_data, add tag */
    if (strcmp(run->data, reference_data) > 0) /* RW data */
        *(int *) run->data += run->tag; /* RW data += Rd tag */
    run = run->next; /* Rd next */
}
...

```

Figure 8.5.: C code of the Linked List application (16 bytes per element)

Element size [bytes]	Processing [clocks]	Total execution [clocks]			Predictor accuracy		
		(Non-)Spec	Spec+Pref	Accel	Rd next	Rd tag	RW data
16	2	10,636,571	10,284,578	1.03x	99.99%	50.16%	99.99%
16	8	11,422,344	10,475,250	1.09x	99.99%	50.16%	99.99%
16	16	12,483,244	10,408,652	1.20x	99.99%	50.16%	99.99%
64	2	12,423,960	12,053,946	1.03x	99.99%	50.00%	99.99%
64	8	13,229,248	12,508,405	1.06x	99.99%	50.00%	99.99%
64	16	14,270,478	12,514,726	1.14x	99.99%	50.00%	99.99%
128	2	14,188,207	13,793,206	1.03x	99.99%	50.00%	99.99%
128	8	15,026,596	14,537,306	1.03x	99.99%	50.00%	99.99%
128	16	16,050,384	14,612,571	1.10x	99.99%	49.99%	99.99%

Table 8.7.: Runtimes and predictor accuracies of the Linked List application (random list layout, 50% writes)

Element size [bytes]	Processing [clocks]	Total execution [clocks]			Predictor accuracy		
		(Non-)Spec	Spec+Pref	Accel	Rd next	Rd tag	RW data
16	2	12,538,664	11,925,887	1.05x	99.99%	73.72%	99.99%
16	8	13,312,916	12,053,269	1.10x	99.99%	73.72%	99.99%
16	16	14,407,643	12,010,401	1.20x	99.99%	73.72%	99.99%
64	2	15,450,386	14,708,844	1.05x	99.99%	73.70%	99.99%
64	8	16,236,104	14,890,484	1.09x	99.99%	73.70%	99.99%
64	16	17,281,926	14,875,836	1.16x	99.99%	73.70%	99.99%
128	2	18,331,562	17,468,961	1.05x	99.99%	73.70%	99.99%
128	8	19,120,235	17,735,062	1.08x	99.99%	73.70%	99.99%
128	16	20,130,528	17,730,822	1.14x	99.99%	73.70%	99.99%

Table 8.8.: Runtimes and predictor accuracies of the Linked List application (random list layout, 80% writes)

Element size [bytes]	Processing [clocks]	Total execution [clocks]			Predictor accuracy		
		(Non-)Spec	Spec+Pref	Accel	Rd next	Rd tag	RW data
16	2	8,755,555	8,563,410	1.02x	99.99%	73.62%	99.99%
16	8	9,550,492	8,683,787	1.10x	99.99%	73.62%	99.99%
16	16	10,580,976	8,678,389	1.22x	99.99%	73.62%	99.99%
64	2	9,468,929	9,330,472	1.01x	99.99%	73.64%	99.99%
64	8	10,264,873	10,085,278	1.02x	99.99%	73.64%	99.99%
64	16	11,302,774	10,129,769	1.12x	99.99%	73.64%	99.99%
128	2	10,184,078	10,060,236	1.01x	99.99%	73.64%	99.99%
128	8	10,982,486	11,483,001	0.96x	99.99%	73.64%	99.99%
128	16	12,024,426	11,576,028	1.04x	99.99%	73.64%	99.99%

Table 8.9.: Runtimes and predictor accuracies of the Linked List application (random list layout, 20% writes)

Element size [bytes]	Processing [clocks]	Total execution [clocks]			Predictor accuracy		
		(Non-)Spec	Spec+Pref	Accel	Rd next	Rd tag	RW data
16	2	2,216,144	2,164,049	1.02x	99.99%	49.88%	99.99%
16	8	3,004,422	2,829,326	1.06x	99.99%	49.88%	99.99%
16	16	4,053,233	3,737,541	1.08x	99.99%	49.88%	99.99%
64	2	6,404,215	6,295,975	1.02x	99.99%	49.88%	99.99%
64	8	7,196,003	7,052,686	1.02x	99.99%	49.88%	99.99%
64	16	8,224,274	7,467,478	1.10x	99.99%	49.88%	99.99%
128	2	10,678,362	10,454,121	1.02x	99.99%	49.88%	99.99%
128	8	11,487,366	11,175,244	1.03x	99.99%	49.88%	99.99%
128	16	12,527,272	11,312,460	1.11x	99.99%	49.88%	99.99%

Table 8.10.: Runtimes and predictor accuracies of the Linked List application (sequential list layout, 50% writes)

8. Experimental Results

cacheline is thus more likely to have been evicted before the remaining adjacent elements are accessed. The severely degraded cache performance observed with large element sizes (especially with the sequential layout, compare execution time ratio for element sizes 16 and 128 in Table 8.10 versus Tables 8.7–8.9) cannot be fully compensated by speculation. However, speculation still achieves speed-ups in all instances except one, the latter can be attributed to excessive cache pollution and thrashing due to aggressive prefetching under adverse access patterns.

Besides the raw cache performance, *speculation accuracy* affects speculation gains as well. To evaluate the speculation accuracy (speculation coverage is 100% by application design), the *predictor accuracy* was measured as

$$\text{predictor accuracy} = \frac{\sum_{i=0}^n x_{\text{Taken},i} P(\text{Taken}_i) + (1 - x_{\text{Taken},i})(1 - P(\text{Taken}_i))}{n}, i, n \in \mathbb{N}$$

with $P(\text{Taken}_i) \in [0, 1]$ being the predicted probability of the i -th memory access being taken, $x_{\text{Taken},i} \in \{0, 1\}$ being the elementary event of the i -th real result recorded afterwards ($x = 1$ taken, $x = 0$ not taken), and n being the total number of speculative memory accesses. In the Linked List benchmark, the *Rd next* and *RW data* ports exhibit near perfect prediction due to static port behavior (accesses are always taken). The prediction accuracy of port *Rd tag* depends on the write ratio, since the read data is only needed if processed and written back afterwards. A 50% write ratio produces the minimum for the prediction accuracy of roughly 50%, as the write probability is distributed equally and no pattern can be recognized by the GAg predictor. When the write ratio is biased toward either direction, the predictor is more likely to recognize patterns of writes repeatedly taken or not taken, respectively. Thus, prediction accuracy for *Rd tag* increases to 74% for both 20% and 80% write ratios. This higher accuracy is also reflected by the greater speed-ups in Table 8.8, with respect to Table 8.7. At 20% write ratio, however, the good cache performance generally limits speculation gains in most instances (Table 8.9).

8.3.2. Tree Search

Tree Search, the second micro-benchmark (C code shown in Figure 8.6), searches for random keys in a 131,072-element binary tree. It uses two read ports, simultaneously speculatively reading both the left and right successors of a node (shown as *Read left* and *Read right* in Table 8.11). In addition, both ports are allocated to a second memory access each, reading the pointers to the successors. The actual control condition is assumed to require from four to 16 clock cycles for evaluation (e.g., a string comparison). The speculation allows the hiding of the main memory latencies, but is limited to just 1.06x improvement (*Accel*) when searching for random keys in the same order as they have been inserted to establish the tree. For short evaluation times of the control condition (*Processing*), speculative prefetching results in a slow-down (*Accel* < 1). This slow-down can be attributed to misspeculated prefetches resulting in cache pollution and memory

```

...
struct tree {
    char string[8];
    struct tree *left;
    struct tree *right;
};

/* Search tree for search_string starting at root */
struct tree *search(struct tree *root, char *search_string) {
    struct tree *run = root;

    while (run) {
        int comparison = strcmp(search_string, run->string);
        if (comparison < 0)
            run = run->left; /* Read left */
        else if (comparison > 0)
            run = run->right; /* Read right */
        else
            /* comparison == 0, found */
            break;
    }

    return run;
}
...

```

Figure 8.6.: C code of the Tree Search application (16 bytes per element)

Element size [bytes]	Processing [clocks]	Total execution [clocks]			Predictor accuracy	
		(Non-)Spec	Spec+Pref	Accel	Read left	Read right
16	4	121,951,897	126,009,925	0.97x	74.27%	72.54%
16	8	134,153,854	129,528,530	1.04x	73.57%	71.92%
16	16	155,628,591	153,414,045	1.01x	77.65%	76.36%
64	4	143,833,487	149,612,097	0.96x	74.46%	72.30%
64	8	156,432,612	151,169,356	1.03x	73.03%	72.11%
64	16	177,773,621	173,298,394	1.03x	77.81%	74.71%
128	4	158,214,158	160,994,051	0.98x	74.02%	72.09%
128	8	172,673,731	162,367,952	1.06x	73.03%	72.11%
128	16	192,309,865	183,020,753	1.05x	76.78%	73.41%

Table 8.11.: Runtimes and predictor accuracies of the Tree Search application (keys in insertion order)

8. Experimental Results

Element size [bytes]	Processing [clocks]	Total execution [clocks]			Predictor accuracy	
		(Non-)Spec	Spec+Pref	Accel	Read left	Read right
16	4	32,946,916	39,856,715	0.83x	79.86%	77.28%
16	8	44,354,741	43,534,493	1.02x	78.10%	75.56%
16	16	66,751,303	114,001,300	0.59x	85.92%	84.83%
64	4	36,043,894	44,349,305	0.81x	80.86%	77.90%
64	8	47,475,613	46,576,374	1.02x	78.68%	76.28%
64	16	69,865,480	128,872,769	0.54x	86.73%	84.97%
128	4	37,938,674	47,158,388	0.80x	81.33%	78.19%
128	8	49,753,062	48,584,735	1.02x	79.07%	76.62%
128	16	71,948,159	134,601,518	0.53x	86.82%	84.84%

Table 8.12.: Runtimes and predictor accuracies of the Tree Search application (keys in ascending order)

bus contention. On the other hand, the short evaluation times limit the per-access gains of successful prefetches. Thus, the adverse effects of the relatively few misspeculated cacheline loads outweigh the benefits of the predominating successful prefetches, despite satisfactory predictor accuracy. A larger cache with smaller cacheline transfer times (e.g., MARC II [LaWK11]) eliminates this disadvantage. However, if such a cache is not feasible due to resource limitations, speculative prefetching should not be applied. Note that pure access speculation (*(Non-)Spec*) is not affected by cache pollution or thrashing (but it does not achieve any speed-ups either).

Predictor accuracies (measured as described above) further improve when *ordered* sequences of keys are being searched for (shown in Table 8.12), indicating that the GAg predictors (described in Section 7.3.2.1) precompute the correct path through the tree and arrange the prioritization of the correct memory accesses. Unfortunately, this does not always result in the desired speed-ups, since pure cache performance without speculation also improves due to the increased memory access locality. When speculative prefetching is applied in addition, cache performance often declines much more pronouncedly due to cache pollution than the speculative prefetching can create benefits. This effect is especially obvious with longer processing times, when early speculation leads to prefetching well in advance, which exceeds cache capacity and evicts cachelines before they can be read again. Again, this could be countered by a larger cache, or by not performing speculative prefetching for affected configurations. Even under these adverse circumstances, however, speculative prefetching achieves speed-ups (*Accel* in Table 8.12) for medium evaluation times of the control condition (*Processing*).

8.3.3. Merge Sort

The third micro-benchmark *Merge Sort* is the inner loop of the CoreMark MergeSort benchmark [EEMB11], which sorts a list of 139,807 elements by merging the sorted sublists in-place. This control-intensive code profits more from speculation, specifically from dynamic prefetching, which again hides the four to 16 cycles latency of the element-wise


```

...
list_head *p, *q, *e, *list, *tail;
signed int psize, qsize;
...
/* Merge lists p and q */
while (psize > 0 || (qsize > 0 && q)) {

    /* Determine next element to merge */
    if (psize == 0) {
        // p is empty, take e from q
        e = q; q = q->next; qsize--; /* Read q */

    } else if (qsize == 0 || !q) {
        // q is empty, take e from p
        e = p; p = p->next; psize--; /* Read p */

    } else if (compare(p->info, q->info) <= 0) {
        // Head of p <= q, take e from p
        e = p; p = p->next; psize--; /* Read p */

    } else {
        // Head of q < p, take e from q
        e = q; q = q->next; qsize--; /* Read q */
    }

    /* Add new element to merged list */
    if (tail) {
        tail->next = e; /* Write tail */
    } else {
        list = e;
    }
    tail = e;
}
...

```

Figure 8.7.: C code of the Merge Sort application [EEMB11]

comparison (shown as function `compare` in Figure 8.7). The element size is fixed at eight bytes, since it was not possible to exercise alternative element sizes without major changes to the MergeSort benchmark. The C code was then manually translated into an HA and executed with two memory configurations.

The first configuration (shown in Table 8.13) flushes MARC’s cache after every HA invocation, which corresponds to one complete set of inner loop iterations. Thus, the

8. Experimental Results

Processing [clocks]	Total execution [clocks]			Predictor accuracy		
	(Non-)Spec	Spec+Pref	Accel	Read p	Read q	Write tail
4	83,223,099	81,325,354	1.02x	99.45%	99.65%	100.00%
8	91,069,866	87,337,745	1.04x	99.46%	99.66%	100.00%
16	104,134,220	99,034,891	1.05x	99.44%	99.65%	100.00%

Table 8.13.: Runtimes and predictor accuracies of the Merge Sort application (cache flushes)

Processing [clocks]	Total execution [clocks]			Predictor accuracy		
	(Non-)Spec	Spec+Pref	Accel	Read p	Read q	Write tail
4	72,242,186	70,384,127	1.03x	99.38%	99.66%	100.00%
8	80,224,839	76,465,413	1.05x	99.40%	99.67%	100.00%
16	93,868,702	88,564,002	1.06x	99.38%	99.66%	100.00%

Table 8.14.: Runtimes and predictor accuracies of the Merge Sort application (shared cache)

Processing [clocks]	Total execution [clocks]			Predictor accuracy		
	(Non-)Spec	Spec+Pref	Accel	Read p	Read q	Write tail
4	57,981,872	56,123,813	1.03x	99.38%	99.66%	100.00%
8	65,964,525	62,205,099	1.06x	99.40%	99.67%	100.00%
16	79,608,388	74,303,688	1.07x	99.38%	99.66%	100.00%

Table 8.15.: Runtimes and predictor accuracies of the Merge Sort application (shared cache, without live variable transfer overhead)

GPP running the SW part of the algorithm can transparently access and manipulate the shared data structures. For all element processing times (*Processing*), speculative prefetching attains speed-ups (*Accel*) of up to 1.05x. The second configuration (shown in Table 8.14) assumes a shared-cache memory architecture, e.g., by connecting both the GPP and HA to MARC and PHASE/V (described in Section 4.6.3). Here, the GPP reads the results directly from the shared cache. Hence, the cache does not have to be flushed after each HA invocation, which results in total savings of ≈ 11 million clock cycles overhead. Thus, speculation gains improve slightly to a maximum of 1.06x. For an even better assessment of pure speculation effects not blurred by system-specific data transfer overheads, the transfer time for live variables (t_{live}) between GPP and HA was measured to be 102 clock cycles per invocation, totalling to

$$t_{live_total} = t_{live} \times n_{elements} = 102 \text{ clocks} \times 139,807 = 14,260,314 \text{ clocks}$$

Table 8.15 shows the results for the shared cache configuration without live variable transfers (subtracted from Table 8.14), which now reflects raw computation time including cache and speculative memory system performance. Consequently, speculation gains improve again with respect to non-speculative (*(Non-)Spec*) execution times.

The results also show very high predictor accuracies (*Read p*, *Read q*, *Write tail*) due to the regularity of the inner loop, which again could be even better exploited [LaWK11] using a larger cache (e.g., MARC II). Nevertheless, the resulting high speculation accuracy enables speed-ups for all memory configurations and element processing times, and thus prevents losses due to cache pollution or memory bus contention as seen with the Tree Search benchmark.

8.3.4. Comrade Sample

Finally, the fourth micro-benchmark *Comrade Sample* sums values from two 100-element arrays: Every third number is taken from one array, the two others from the second array. This benchmark has actually been automatically compiled by Comrade (see Section 6.5.1) from C (source code shown in Figure 8.8) into an HA, which was then manually fitted with the speculative memory system. Comrade already supports powerful speculation mechanisms within the generated HA datapath, but currently targets pure MARC (without memory speculation), thus the need for manual intervention. Again, element sizes were varied from four to 128 bytes. However, the per-element processing time is fixed by the application-specific Comrade datapath.

Speculation gains (shown as *Accel* in Table 8.16) increase from moderate 1.01–1.02x for small element sizes to significant 1.65x for 128-byte elements. In contrast to the first three benchmarks examined in this section, which only achieve relevant speed-

```

...
int i, t;
int *p, *q;
...

/* Sum 100 array elements */
t = 0;
for (i = 0; i < 100; i++) {
    /* Take every 3rd number from p ... */
    if ((i & 3) == 0) {
        t += *p + 1; /* Read p */
        /* ... otherwise from q */
    } else {
        t += *q + 2; /* Read q */
    }
    p++;
    q++;
}
...

```

Figure 8.8.: C source code of the Comrade Sample application

8. Experimental Results

Element size [bytes]	Total execution [clocks]					Predictor accuracy	
	Non-spec	Spec	Accel	Spec+Pref	Accel	Read p	Read q
4	823	811	1.01x	804	1.02x	98.25%	94.04%
32	3,035	2,896	1.05x	2,903	1.05x	97.59%	96.41%
64	5,602	5,426	1.03x	5,421	1.03x	96.49%	97.36%
128	10,336	6,280	1.65x	6,255	1.65x	85.19%	94.81%

Table 8.16.: Runtimes and predictor accuracies of the Comrade Sample application

ups by combined access prediction and speculative prefetching, Comrade’s advanced speculation mechanisms attain most of the total speed-up by employing access prediction/prioritization *only*, due to the tight integration of Comrade’s datapath controller with the speculative memory system. Moreover, the highly efficient access prioritization can be attributed to the extreme case of highly regular *alternating* access patterns used in this benchmark. Thus, it is hardly surprising that the predictor accuracy shines as well, ranging from 85% to 98%. On the other hand, large strides as seen with 128-byte elements degrade cache performance, while not providing much potential for prefetching (every cacheline holds just a single element). Hence, speculative prefetching merely improves over access prediction.

8.3.5. Implementation Aspects

In summary, the type of memory-based speculation that should be employed (if any) strongly depends on the nature of the application. The speed-up achievable by speculation also depends on cache performance, since the cache cannot be outperformed (discussed in the previous Sections 8.3.1–8.3.4). Similar to other prefetching schemes, speculative prefetching is only profitable if the next memory access does not follow in direct sequence. Instead, prefetching hides latencies, e.g., due to address calculation or data processing. However, too aggressive speculative prefetching can provoke cache pollution, adversely affecting overall performance.

Fortunately, very little hardware resources are required to integrate lightweight, but powerful speculation mechanisms into the Reconfigurable Computing Platform. Table 8.17 shows the mapping results for the rSoC containing each of the four benchmarks

Application	Non-spec		Spec		Spec+Pref		Block RAM
	4-LUTs	FFs	4-LUTs	FFs	4-LUTs	FFs	
Linked List	15,838	7,858	16,289	8,011	16,166	8,122	32
Tree Search	12,226	7,372	12,520	7,222	12,623	7,363	32
Merge Sort	16,078	7,923	16,558	8,003	16,560	8,175	32
Comrade Sample	12,688	7,849	13,084	8,096	13,104	8,101	32

Table 8.17.: FPGA areas for rSoCs with specified application and memory-based speculation

presented in this section. The rSoC (including MARC and FastLane+) was synthesized using Synplify 9.6.2 and subsequently mapped with ISE 10.1.03. Each application was configured without speculation, just access prediction, and full speculative prefetching (including access prediction) enabled in HW. Due to its low area overhead (typically less than 4%), the lightweight speculative memory system can always be included in the Reconfigurable Computing Platform, and speculation be later enabled at runtime on a per-application basis.

8.4. Power and Energy Consumption

Since FPGAs are often used in low-power embedded environments, the power impact of the speculative memory system as well as the FastPath and FastLane+ techniques has also been evaluated. To this end, combinations of the pointer chasing HA traversing a linked list of 128K elements (described in Section 8.1.3) and SW programs running on the GPP were exercised on the Xilinx ML310 board, and the board-level supply currents were measured at the ATX (Advanced Technology Extended) power connector. The ML310 (described in Section 4.3) only uses the +5 V, +12 V, and -12 V ATX rails, the +3.3 V and -5 V rails are not connected. The main supplies feeding the Virtex-II Pro FPGA, the DDR SDRAM, and other on-board devices are derived from the +5V rail [Xili05a], while the remaining voltages are used for peripheral I/O.

Table 8.18 shows the board-level supply currents and total power consumption for the pointer chasing HA running in parallel with the GSM and pointer chasing SW applications (described in Section 8.1.2), both executed on the GPP using the AISLE program layout (Section 4.6.2). The cases of the SW system or the HA idling were also considered. The measurements show that the active power consumption of the accelerators including FastPath/FastLane+ pales in comparison to that of the rest of the system (e.g., PowerPC GPP, network interfaces, memories, etc.). The peak difference between system idle power and power consumed with both GPP and accelerator under full load is less than 14%. This does not come as a surprise, since the new hardware components encompass at most a few hundred LUTs and Flip-Flops, the entire rest of the rSoC remains unchanged. Hence, the results can be interpreted as representative

HW accelerator	GPP software	Supply current [A]			Total power [W]
		+5 V	+12 V	-12 V	
Idle	Idle system	1.717	0.090	0.025	9.97
Idle	Pointer chasing	1.788	0.090	0.025	10.32
Idle	GSM	1.774	0.090	0.025	10.25
Pointer chasing	Idle system	1.938	0.090	0.025	11.07
Pointer chasing	Pointer chasing	1.983	0.090	0.025	11.30
Pointer chasing	GSM	1.990	0.090	0.025	11.33

Table 8.18.: ML310 board-level supply currents and total power consumption for specified combinations of hardware accelerator and SW application

8. Experimental Results

Pointer chasing	Total power* [W]	Execution time [†] [ms]	Total energy [J]
GPP software	10.32	68.3	0.70
HW accelerator	11.07	27.4	0.30

*taken from Table 8.18

[†]taken from Table 8.4

Table 8.19.: Total energy consumption for the pointer chasing application executed in SW and as HW accelerator

of all HA-SW combinations examined in this work, including the speculative memory system (cf. small area overhead for speculation in Table 8.17).

Of course, total *energy* consumption is significantly reduced by the FastPath/FastLane+ combination since it achieves much shorter hardware and software execution times (see Table 8.2, compare with Table 8.3). Table 8.19 shows the total energy consumption of the pointer chasing application running as HA and in SW on the GPP, respectively. While the total power consumption for the HA is slightly higher than that of the SW version, the HA version reduces energy consumption to only 43%, since it requires less than half the execution time of the SW version. Likewise, longer or shorter hardware execution times increase or reduce the energy consumption of the speculative memory system. Whenever memory speculation saves on power-intensive memory accesses and shortens hardware execution times (see Tables 8.7–8.16), it also conserves energy.

9. Summary and Future Work

In an effort to improve the performance, ease of design, and re-usability of embedded systems and their building blocks, this work introduced the Reconfigurable Computing Platform, a novel approach to design high-performance flexible systems that leverage an efficient architecture based on a speculative high-bandwidth memory system and a low-latency execution model, both tailored specifically to application requirements. The Platform is either composed from components that are automatically generated by an integrated hardware/software compile flow, or manually assembled from intellectual property cores designed to application characteristics. The properties of both hardware and software components, the interfaces between them, and the execution model are expressed in a single consistent platform description language and management framework.

After describing the limitations of current design techniques, execution models, and reconfigurable computing architectures, especially with regard to automatic hardware/software compilation and efficient interfacing between the resulting hardware accelerators and software parts of an application, four Columns were identified that support the design of Reconfigurable Computing Platforms. Each Column raised research questions, that were elaborated in the four corresponding main chapters of this work, and that will now be revisited for summarizing the findings of this research.

What characteristics does an execution model for Reconfigurable Computing Platforms encompass? How can they be met in practice on a real system?

As the First Column supporting the Reconfigurable Computing Platform that this thesis aims to realize, a novel model of execution was introduced, orchestrating the interaction between a conventional software programmable processor and hardware accelerators. Next, it was shown how to efficiently realize this model on actual hardware.

In the aim to demonstrate the practical feasibility of reconfigurable computing systems supporting fine-grained application partitioning between general purpose processor(s) and hardware accelerator(s), the potential of reconfigurable computing systems for accelerating non-streaming, pointer-chasing code over software versions was shown. One speed advantage of modern general purpose processors is often due to the tight integration of fast multi-megabyte caches within the processor, something generally not possible with commercially available reconfigurable devices. However, the moment the size of irregular data sets exceeds the cache size (such as for railway routing graphs [MüSc04]), processor performance drops to the speed of the memory system. Modern reconfigurable devices *are* already reaching these speeds, but beyond that can then exploit increased parallelism, both with regard to number of memory banks and processing elements.

To support such a highly parallel execution model, a high performance memory

9. Summary and Future Work

attachment for custom hardware accelerators was presented. This approach can increase the usable memory throughput by more than 8x over the original vendor-provided PLB attachment (included in the Xilinx EDK [Xili06] design suite), almost reaching the theoretical throughput of the memory chips themselves (and significantly exceeding that of the general purpose processor). Additionally, it required less chip area and left performance of software applications running on the on-chip processors almost unaffected.

From a practical perspective, FastLane+ integrates into the standard EDK design flow and is as easy to use as the original attachment. Although the results are not directly transferable to platforms other than Virtex-II Pro/Virtex-5 FX FPGAs, it is clear that reduced bus and wrapper overhead will always result in smaller logic and lower latencies. Hence, other platforms may also benefit from this approach.

Beyond the memory access itself, it was also further examined how a general purpose processor and a hardware accelerator can interact in a virtual memory environment. With the new PHASE/V technology, it was shown that full demand paging *is* possible even if the memory management unit of the general purpose processor is not directly accessible to the hardware accelerator. Apart from memory throughput, the effective speed-up achievable for a finely-partitioned application is also highly dependent on the delay of inter-partition communications. Using the new FastPath low-latency communication mechanism, even shorter fragments of application code can now be successfully hardware-accelerated.

What is platform management? Why is it important for automatic HW/SW compilers in particular?

To maintain and enhance platform building block portability and compatibility in a heterogeneous and evolving hardware/software design environment, an abstract and flexible representation of design properties and parameters for the increasingly complex platform-based embedded systems is required. Moreover, a well-defined target platform was found to be a prerequisite for automatic hardware/software compilation and electronic system level design tools. Hence, a platform and configuration management system was presented as the Second Column to model and efficiently handle the high-performance Reconfigurable Computing Platform.

Although not part of the Platform's run-time logic, support for automated platform management proved essential for even medium-scale platforms, which already comprise many heterogeneous components, buses, interfaces and large parameter sets. To this end, the CoMAP repository was introduced to hold the definitions for interfaces, components and their parameters, and platforms. Mechanisms to automatically connect interfaces and buses to compose a working platform were developed. The repository uses data models and representations based on the study of more than thirty commercial intellectual property (IP) cores, that were classified using the attributes of the CoMAP interface template. All of the IP cores' interface semantics could be successfully described with the existing attribute catalog.

From a practical perspective, support for tool-assisted preparation and automated validation of complex parameter sets and interdependencies was identified as a key to

automatic platform management. Hence, tool support was introduced to assist the system designer with extracting, processing, preparing, and inserting repository elements as well as relieving parameter-related limitations of common EDA tools. However, direct manipulation of repository data is still possible due to a human-readable notation.

The effectiveness of both platform description and composition was shown with a real world example that demonstrated the interoperability with the other parts of the Reconfigurable Computing Platform by modeling a commercially available rSoC.

What are the characteristics of the HW/SW interface? How can such an interface be made available to automatic HW/SW compilation and platform composition without restricting its functionality?

Embedded reconfigurable systems-on-chip, platform-based designs, and automatic hardware/software compilation flows require, or can benefit from, integration of configurable hand-optimized intellectual property (IP) cores from multiple sources. PaCIFIC represents the Third Column supporting Reconfigurable Computing Platforms, a strategy for using complex intellectual property cores from within ANSI C programs as seamlessly as pure C software functions. PaCIFIC encapsulates the hardware specifics unfamiliar to a software developer and guarantees that the interplay of hardware and software as well as the data exchange between them adhere to the execution model and its underlying architectures developed in this work. The intellectual property provider supplies the details required for core integration as a human-readable formal description. The hardware/software interfaces are then generated automatically, thus raising design productivity by closing the gap between the vertical hardware and software design flows.

To this end, a system and interface communication model was developed based on the evaluation of many commercial IP cores, which also exploits the detailed hardware core and interface definition features of CoMAP. For the hardware interface, two classes of properties were identified to define static as well as dynamic interface behavior. Two corresponding interface types were then designed for the software side, which provide for automatic simple interface generation and encapsulation of complex IP core functions in a single function call respectively, while maintaining a natural C programming style.

This approach applies not only to Comrade or the specific domain of adaptive computing systems, but generally to all hardware/software co-design environments. Reusable interface descriptions allow the separation of interfaces and implementation details. Additionally, Reconfigurable Computing Platforms such as adaptive computers profit from PaCIFIC's ability to generate lightweight native interfaces at the datapath-level between intellectual property cores and the rest of the system. This avoids the overhead incurred by requiring on-chip bus-compatible wrappers between the generic hardware blocks.

The realization of this combined hardware/software approach was demonstrated using Comrade as the host hardware/software compiler to integrate the additional compiler passes for the new PaCIFIC mechanisms. Finally, a real world example is shown, which uses conventional C language to seamlessly integrate a standard IP core with the Reconfigurable Computing Platform that is defined by this work.

Why is speculative program execution especially effective in the context of hardware-accelerated computing? Does an efficient speculative memory system always have to be costly?

A parameterized lightweight speculative memory system was presented, which is suitable for platform-based (reconfigurable) system-on-chip designs as well as target for automatic high-level language to hardware compilers for reconfigurable computers. Next, an efficient implementation was shown which saves on resources by reusing and interacting with the existing MARC system, and which integrates well as the Fourth Column with the other parts of this work that constitute the Reconfigurable Computing Platform.

To this end, the main aspects of speculative program execution were identified. Control, data, and address speculation as well as prediction were then examined for their suitability for single and multi-core architectures, the latter with special regard to parallel execution models for hardware accelerators. In addition, for parallel hardware accelerator execution it proved advantageous to directly support speculative execution at the memory system level, avoiding inefficiencies incurred when considering it only at the computation level.

The system fully supports the spatial computation execution model by allowing the realization of each memory operator by a dedicated hardware memory port. Efficient speculative execution is enabled by a dynamic scheme for arbitrating access to shared resources such as main memory, relying on techniques inspired by the branch prediction of conventional software-programmable processors. It is the first system to combine support for control speculation and distributed memories with the FastLane+ system interface to general purpose processor and main memory. Thus, the speculative memory system significantly increases the exploitation efficiency of the high memory bandwidth provided by FastLane+.

The efficacy of the speculative memory system was then shown by evaluating a number of hardware-accelerated micro-benchmarks and base algorithms with several combinations of memory-based control speculation and speculative prefetching, demonstrating speed-ups of up to 1.65x.

9.1. Lessons Learned

While working on and experimenting with the Reconfigurable Computing Platform, it was found insufficient to merely consider and improve an isolated part of the system, since other components of the Platform cannot handle the higher performance of the now enhanced single aspect and thus become bottlenecks themselves. Although many of the underlying concepts have already been explored separately, it is their combination that catalyzes a high-performance efficient Reconfigurable Computing Platform with a matching and easy-to-use hardware/software co-design flow. For instance, the speculative memory system can only realize its capabilities in conjunction with the FastLane+ high-performance memory attachment and the advanced signalling of the execution

model. Likewise, it is the unified notation for intellectual property configuration and interface protocol description that enables (semi-) automatic design composition as well as automatic hardware/software compilation. Hence, the combined Four Column approach proved to be more than the sum of its individual measures.

9.2. Future Research

Due to the ever-growing complexity of embedded systems, an increasing need for comprehensive approaches and solutions for Reconfigurable Computing Platforms can be expected. One area of future work is increasing the efficiency of the execution model. Even using FastPath, the explicit interrupt request for the general purpose processor to resolve a hardware TLB page fault does have a relatively high overhead. While the general purpose processor needs typically 210 ns (min. 60 ns, max. 1710 ns) per page table walk to keep its TLB up to date, the interrupt latency using FastPath is still around an order of magnitude higher ($2.1\ \mu\text{s}$ – $8.8\ \mu\text{s}$). This could be avoided by having a shared TLB between general purpose processor and hardware accelerator, which can be implemented in reconfigurable logic at full bus speeds as demonstrated in this thesis. The processor-internal TLB could then be switched off and the hardware TLB be accessed instead without additional performance penalties. Other possible areas of future research include the simultaneous sharing of the reconfigurable compute unit between multiple different software processes.

In support of the execution model and the speculative memory system, the interface between general purpose processor and hardware accelerator should be improved. One such interface candidate is HyperTransport, a well-known proprietary yet quasi-open standard. Previous work [SGNB08] attached a hardware accelerator using HTX, the system-bus subset of HyperTransport. However, the limited HTX interface performance could be increased by using the full-featured HyperTransport CPU socket [Hype05] instead, which could reach full HyperTransport speed in conjunction with the new multi-gigabit serial transceivers available in most modern FPGAs.

Work is under way to create MARC II [LaWK11], which extends MARC by integrating the speculative memory system and introducing cache coherency clusters as well as local improvements (e.g., low-degree associativity, victim line handling, etc.). The potential gains of write data speculation come at hardware resource costs prohibitively high for current devices, but might become feasible for future generations of reconfigurable technology. Hence, it is also planned to extend the support for speculative writes beyond write prefetching and provide full support for multiple speculative versions of write data in flight. Value speculation such as prediction of live-in variable values, memory data, and addresses is expected to further increase speculation benefits.

Accordingly, the automatic hardware/software compiler Comrade is enhanced to target MARC II and automatically configure it as appropriate for the current application (e.g., assignment of memory ports, select speculation and coherence strategy, etc.). Furthermore, tool support for intellectual property core integration using PaCIFIC should be improved within Comrade. To this end, Comrade could be augmented with

9. Summary and Future Work

a new back end that interfaces to existing commercial tools (e.g., Xilinx EDK). By generating platform descriptions that integrate the compiled custom datapath and the intellectual property cores with the Reconfigurable Computing Platform, the compiler would be able to leverage the rich intellectual property libraries and design tool support of commercial (reconfigurable) system-on-chip platforms. In the process, the CoMAP repository, being an integral part of the PaCIFIC framework, would have to be combined with commercial platform/component libraries and their associated tools as well. CoMAP would thus form a meta-interface for Comrade, which would still be able to target a well-defined Reconfigurable Computing Platform.

The techniques presented so far also did not consider the capability to dynamically reconfigure an FPGA, which is now becoming sufficiently reliable to be supported in industrial design flows. Work has already started to support it both in the model as well as the hardware prototype.

To conclude, the comprehensive approach to improve the Reconfigurable Computing Platform by tackling both platform organization and communication issues on both architectural as well as technical levels proved successful, since most design and performance bottlenecks could be eliminated or alleviated. Due to its broad focus, the project also yielded many ideas as well as new areas and directions for future work. Among these, it is especially recommended to establish a unified hardware/software operating system as continuous abstraction layer, that provides system services for the Reconfigurable Computing Platforms via a complete hardware/software application programming interface. Thus, the reusable and well-defined Platform would further facilitate application development and interaction with existing electronic system level design tools.

Bibliography

- [Acte10a] Actel Corp., “Cortex-M1 v3.0 Handbook”, *Actel 50200127-11*, 2010
- [Acte10b] Actel Corp., “Actel SmartFusion Microcontroller Subsystem User’s Guide”, *Actel 50200250-1*, 2010
- [Acte10c] Actel Corp., “IGLOO FPGA Fabric User’s Guide”, *Actel 50200257-0*, 2010
- [Acte10d] Actel Corp., “Solutions and IP Catalog”, *Actel 51900036-15/3.10*, 2010
- [AEGH10] Arnesen A., Ellsworth K., Gibelyou D., Haroldsen T. et al., “Increasing Design Productivity Through Core Reuse, Meta-Data Encapsulation, and Synthesis”, Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL), Italy, 2010
- [Agil09] Agility Design Solutions Inc. / Mentor Graphics Corporation, “Handel-C Language Reference Manual”, 2009
- [Alte02] Altera Corp., “Excalibur Device Overview”, *Altera DS-EXCARM-2.0*, 2002
- [Alte09] Altera Corp., “Megafunction Overview User Guide”, *Altera UG-01056-1.0*, 2009
- [Alte10] Altera Corp., “Nios II Processor Reference Handbook”, *Altera NII5V1-10.0*, 2010
- [AMKS08] André C., Mallet F., Khan A. M., de Simone R., “Modeling SPIRIT IP-XACT with UML MARTE”, Workshop on Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML Profile, Conf. on Design, Automation, and Test in Europe (DATE), Germany, 2008
- [AnGW10] Andersson J., Gaisler J., Weigand R., “Next Generation Multipurpose Microprocessor”, Intl. Conf. on Data Systems in Aerospace (DASIA), Hungary, 2010
- [Appe98] Appel A., “Modern Compiler Implementation in C”, Cambridge University Press, 1998
- [ArDr99] Aron M., Druschel P., “Soft timers: efficient microsecond software timer support for network processing”, ACM SIGOPS Operating Systems Review, Volume 33 , Issue 5, ACM, 12/1999

- [Arml10a] ARM Ltd., “Cortex-A9 Technical Reference Manual”, Revision r2p2, *ARM DDI 0388F (ID050110)*, 2010
- [Arml10b] ARM Ltd., “AMBA AXI Protocol Specification v2.0”, *ARM IHI 0022C*, 2010
- [ATIT05] ATI Technologies Inc., “HyperMemory – Next-Generation memory management for PCI Express graphics”, White Paper, http://www.ati.com/technology/HyperMemory_Whitepaper.pdf, retrieved 04/2008, 2005
- [Auto10] AutoESL Design Technologies Inc., “AutoPilot FPGA High-Level Synthesis”, http://www.autoesl.com/docs/AutoPilot_FPGA_datasheet_oct_2010.pdf, retrieved 10/2010, 2010
- [Bala07] Balacco S., “Linux in the Embedded Systems Market (Vol. VII)”, Venture Development Corp., 2007
- [BaLa96] Ball T., Larus J., “Efficient path profiling”, Proc. 29th ACM/IEEE International Symposium on Microarchitecture, Paris, France, 1996
- [Batc68] Batcher K. E., “Sorting networks and their applications”, AFIPS Conf. Proc. 32, 307–314, 1968
- [BeLT09] Bennis A., Leiser M., Tadmor G., “Implementing a Highly Parameterized Digital PIV System on Reconfigurable Hardware”, Proc. 20th IEEE Intl. Conf. on Application-specific Systems, Architectures and Processors (ASAP), USA, 2009
- [BEMN03] Baumgarte V., Ehlers G., May F., Nückel A. et al., “PACT XPP – A Self-Reconfigurable Data Processing Architecture”, The Journal of Supercomputing, Vol. 26, No. 2, pp. 167–184, Kluwer, 2003
- [Berk10a] Berkeley Design Technology Inc., “An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool”, http://www.autoesl.com/docs/bdti_autopilot_final.pdf, retrieved 10/2010, 2010
- [Berk10b] Berkeley Design Technology Inc., “BDTI Certified Results for the Synfora PICO High-Level Synthesis Tool”, http://www.bdti.com/bdtimark/hlstop_synfora.html, retrieved 10/2010, 2010
- [BGHS08] Bobba J., Goyal N., Hill M. D., Swift M. M., Wood D. A., “TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory”, Proc. 35th Intl. Symp. on Computer Architecture, China, 2008

- [Bill03] Billauer E., “A Guide to PERLilog”, <http://billauer.co.il/download/perlilog-guide-0.3.pdf>, retrieved 10/2010, 2003
- [BILM06] Blundell C., Lewis E. C., Martin M. M. K., “Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions”, Technical Report Nr. CIS-06-09., Department of Computer and Information Science, University of Pennsylvania, 2006
- [BrNe99] Brelet J. L., New B., “Designing Flexible, Fast CAMs with Virtex Family FPGAs”, *Xilinx Application Note XAPP203*, 1999
- [BrSG10] Brandon A., Sourdis I., Gaydadjiev G. N., “General Purpose Computing with Reconfigurable Acceleration”, Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL), Italy, 2010
- [BVCG04] Budiu M., Venkatarami G., Chelcea T., Goldstein S. C., “Spatial Computation”, Proc. Intl. ACM Conf. on ASPLOS, Boston, USA, 2004
- [BWAK08] Burke D., Wawrzynek J., Asanovic K., Krasnov A., Schultz A., Gibeling G., Droz P.Y., “RAMP Blue: Implementation of a Manycore 1008 Processor System”, Proc. Reconfigurable Systems Summer Institute (RSSI), Urbana, USA, 2008
- [Cade08] Cadence Design Systems Inc., “Cadence C-to-Silicon Compiler Delivers on the Promise of High-Level Synthesis”, *Technical Paper 20553 07/08*, 2008
- [CaHW00] Callahan T., Hauser J., Wawrzynek J., “The Garp Architecture and C Compiler”, IEEE Computer, 04/2000
- [CCEK09a] Chaudhry S., Cypher R., Ekman M., Karlsson M. et al., “Rock: A High-Performance Sparc CMT Processor”, IEEE Micro, 29(2) pp. 6–16, 2009
- [CCEK09b] Chaudhry S., Cypher R., Ekman M., Karlsson M. et al., “Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun’s Rock Processor”, Proc. 36th Intl. Symp. on Computer Architecture (ISCA), USA, 2009
- [CFRW91] Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K., “Efficiently computing static single assignment form and the control dependence graph”, ACM Transactions on Programming Languages and Systems, 13(4):451–490, Oct 1991
- [ChHe09] Chiu M., Herbordt M.C., “Efficient Particle-Pair Filtering for Acceleration of Molecular Dynamics Simulation”, Proc. 19th Intl. Conf. on Field Programmable Logic and Applications (FPL), Czech Republic, 2009

- [ChOB92] Chou P., Ortega R., Borriello G., “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”, Proc. Intl. Conf. on Computer Aided Design (ICCAD), USA, 1992
- [ChTJ04] Ching J., Thaker H. M., Jackman S. et al., “VBPP 1.1.0”, <http://svn.seul.org/viewcvs/viewvc.cgi/eda/vbpp/?root=SEUL>, retrieved 10/2010, 2004
- [Clic09] Click C., “Azul’s Experiences with Hardware Transactional Memory”, Bay Area Workshop on Transactional Memory, USA, 2009
- [CNHF08] Chung E.S., Nurvitadhi E., Hoe J. C., Falsafi B., Mai K., “Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations Using FPGAs”, Proc. 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), Monterey, 2008
- [Colw04] Colwell B., “Things CPU Architects Need To Think About”, Stanford Computer Systems Laboratory Colloquium (EE380), Stanford University Department of Electrical Engineering, USA, 02/2004
- [Conv08] Convey Computer Corporation, “The Convey HC-1 Computer”, White Paper, 2008
- [Conv10] Convey Computer Corporation, “Convey’s hybrid-core technology: the HC-1 and the HC-1^{ex}”, *Convey CONV-10-017.1*, 2010
- [Corb08] Corbet J., “Memory management notifiers”, LWN.net, Eklektix Inc., 23/01/2008
- [DaTC09] Davis J.D., Thacker C.P., Chang C., “BEE3: Revitalizing Computer Architecture Research”, Microsoft Research Technical Report no. MSR-TR-2009-45, 2009
- [Davi10] Davis D. L., “Smart, Connected, Transformational – Fueling the Continuum of Computing with Intel Atom Processor”, Keynote, Intel Developer Forum (IDF), USA, 2010
- [Deha10] DeHaven K., “Extensible Processing Platform - Ideal Solution for a Wide Range of Embedded Systems”, *Xilinx WP369*, 2010
- [Dena10] Denali Software Inc., “Blueprint”, <http://www.denali.com/en/products/blueprint.jsp>, retrieved 09/2010, 2010
- [Desi05] Design And Reuse, “Poseidon automates the generation of hardware accelerator modules for Xilinx Virtex-4 FPGAs”, Design & Reuse Headline News, June 13, 2005

- [DHKM04] Danek M., Honzik P., Kadlec J., Matousek R., Pohl Z., “Reconfigurable system-on-a-programmable-chip platform”, Proc. 7th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, Slovakia, 2004
- [DLBT04] Donlin A., Lysaght P., Blodget B., Troeger G., “A Virtual File System for Dynamically Reconfigurable FPGAs”, Proc. 14th Int. Conf. on Field Programmable Logic and Applications (FPL), Antwerp, 2004
- [DoLo04] Donn timer C., Lorenz U., “The Chess Monster Hydra”, Proc. 14th Intl. Conf. on Field-Programmable Logic and Applications (FPL), Belgium, 2004
- [DOSG02] Doucet F., Otsuka M., Shukla S., Gupta R., “An Environment for Dynamic Component Composition for Efficient Co-Design”, Proc. Design Automation and Test in Europe, Paris, France, 2002
- [DoTR01] Dovrolis C., Thayer B., Ramanathan P., “HIP: hybrid interrupt-polling for the network interface”, ACM SIGOPS Operating Systems Review, Volume 35 Issue 4, ACM, 10/2001
- [DuGV04] Dunkels A., Grönvall B., Voigt T., “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”, Proc. IEEE Workshop on Embedded Networked Sensors (Emnets-I), USA, 2004
- [Duo110] Duolog Technologies Ltd., “SOCRATES Chip Integration Platform”, Product Brief, http://www.duolog.com/files/pub/Socrates_hub.pdf, retrieved 10/2010, 2010
- [EbMP09] Ebrahimi E., Mutlu O., Patt Z. N., “Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems”, Proc. 15th Intl. Symp. on High-Performance Computer Architecture (HPCA), USA, 2009
- [Ecli10] Eclipse Foundation Inc., “Eclipse Helios (3.6) Documentation”, <http://www.eclipse.org/documentation>, retrieved 10/2010, 2010
- [EEMB11] Embedded Microprocessor Benchmark Consortium (EEMBC), “What is CoreMark?”, <http://www.coremark.org>, retrieved 01/2011, 2011
- [Euro00] European Telecommunications Standards Institute, “Digital cellular telecommunications system (Phase 2+); ANSI-C code for the GSM Enhanced Full Rate (EFR) speech codec”, *standard ETSI EN 300 724 V8.0.1*, 2000
- [EzJo05] Ezer S., Johnson S., “Smart Diagnostics for Configurable Processor Verification”, Proc. 42nd Design Automation Conference (DAC), USA, 2005
- [FiFY05] Fisher J., Faraboschi P., Young C., “Embedded Computing: A VLIW Approach to Architecture, Compiler and Tools”, chapter 11.1, Elsevier, 2005

- [FMBW09] Fürst S., Mössinger J., Bunzel S., Weber T. et al., “AUTOSAR - A World-wide Standard is on the Road”, 14th Intl. VDI Congress Electronic Systems for Vehicles, Germany, 2009
- [FPRS09] Friedl M., Provos N., de Raadt T., Steves K. et al., “OpenSSH 5.3 Release Notes”, <http://www.openssh.com/txt/release-5.3>, retrieved 10/2010, 2009
- [FrSo96] Franklin M., Sohi G., “ARB: A Hardware Mechanism for Dynamic Reordering of Memory References”, IEEE Transactions on Computers, Vol. 45, No. 5, pp. 552–571, 1996
- [GaCo07] Garcia P., Compton K. A., “Reconfigurable Hardware Interface for a Modern Computing System”, Proc. 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, 2007
- [GaCo09] Garcia P., Compton K. A., “Shared Memory Cache Organizations for Reconfigurable Computing Systems”, Proc. 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, 2009
- [Gädk10] Gädke H., personal communication, 03/2010
- [GäKo08] Gädke H., Koch A., “Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens”, LNCS Intl. Workshop on Applied Reconfigurable Computing, London, UK, 2008
- [GäSK08] Gädke H., Stock F., Koch A., “Memory Access Parallelization in High-Level Language Compilation for Reconfigurable Adaptive Computers”, IEEE Intl. Conf. on Field Programmable Logic and Applications (FPL), Germany, 2008
- [GäTK10] Gädke-Lütjens H., Thielmann B., Koch A., “A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation”, Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL), Italy, 2010
- [GCMG94] Gallagher D., Chen W., Mahlke S., Gyllenhaal J., Hwu W., “Dynamic Memory Disambiguation Using the Memory Conflict Buffer”, Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), USA, 1994
- [GGDN04] Gupta S., Gupta R., Dutt N., Nicolau A., “SPARK – A Parallelizing Approach to the High-Level Synthesis of Digital Circuits”, Kluwer, 2004
- [Giac04] Giacobbi G., “GNU Netcat 0.7.1”, *user manual*, <http://netcat.sourceforge.net/download.php>, retrieved 10/2010, 2004
- [GoGr95] Gokhale M., Graham P.S., “Reconfigurable Computing”, Springer, 2005

- [Goog10] Google Inc., “What is Android?”, Android 2.2 r1, <http://developer.android.com/guide/basics/what-is-android.html>, retrieved 10/2010, 2010
- [GoRo83] Goldberg A., Robson D., “Smalltalk-80: The Language and its Implementation”, Addison-Wesley, 1983
- [GrJN09] Grannaes M., Jahre M., Natvig L., “Storage Efficient Hardware Prefetching Using Delta Correlating Prediction Tables”, Proc. 1st Journal of Instruction-Level Parallelism (JILP) Data Prefetching Championship (DPC), USA, 2009
- [GSAK00] Gokhale M. B., Stone J. M., Arnold J., Kalinowski M., “Stream-oriented FPGA Computing in the Streams-C High-Level Language”, Proc. IEEE Symp. on FCCM, Napa, USA, 2000
- [GSVG10] Goulding N., Sampson J., Venkatesh G., Garcia S. et al., “GreenDroid: A Mobile Application Processor for a Future of Dark Silicon”, Hot Chips - A Symp. on High Performance Chips, USA, 2010
- [GVSS98] Gopal S., Vijaykumar T., Smith J., Sohi G., “Speculative Versioning Cache”, Proc. 4th Intl. Symp. on High-Performance Computer Architecture (HPCA), USA, 1998
- [HaDe08] Hauck S., DeHon A. (Eds.), “Reconfigurable Computing: The Theory and Practice of FPGA-based Computation”, Morgan Kaufmann, 2008
- [HaKR95] Hartenstein R., Kress R., Reinig H., “A Scalable, Parallel, and Reconfigurable Datapath Architecture”, 6th Intl. Symposium on IC Technology, Systems & Applications (ISIC95), Singapore, 1995.
- [Half07] Halfhill T. R., “MicroBlaze v7 Gets an MMU”, Microprocessor Report, November 13, 2007
- [HaWO98] Hammond L., Willey M., Olukotun K., “Data Speculation Support for a Chip Multiprocessor”, Proc. 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), USA, 1998
- [HeMo93] Herlihy M., Moss J. E. B., “Transactional Memory: Architectural Support for Lock-Free Data Structures”, Proc. 20th Annual Intl. Symp. on Computer Architecture (ISCA), USA, 1993
- [HePa06] Hennessy J. L., Patterson D. A., “Computer Architecture: A Quantitative Approach”, 4th Edition, Morgan Kaufmann, 2006
- [HLSK08] Hildenbrand D., Lange H., Stock F., Koch A., “Efficient Inverse Kinematics Algorithm based on Conformal Geometric Algebra Using Reconfigurable Hardware”, Intl. Conf. on Computer Graphics Theory and Applications (GRAPP), Portugal, 2008

- [Hoar85] Hoare T., “Communicating Sequential Processes”, Prentice Hall International Series in Computer Science, 1985
- [Holl10] Holland C., “Embedded boards market grows”, Embedded.com European Newsletter, EE Times Group, 29/07/2010
- [HuKi10] Huong G. Kim S., “Support of cross calls between a microprocessor and FPGA in CPU-FPGA coupling architecture”, Proc. 17th Reconfigurable Architectures Workshop (RAW), USA, 2010
- [HWCC04] Hammond L., Wong V., Chen M., Carstrom B. et al., “Transactional Memory Coherence and Consistency (TCC)”, Proc. 31st Intl. Symp. on Computer Architecture (ISCA), Germany, 2004
- [Hype05] HyperTransport Technology Consortium, “HyperTransport I/O Link Specification Revision 2.00b”, *Document # HTC20031125-0035-0009*, 2005
- [IeFl08] Institute of Electrical and Electronics Engineers, “IEEE Standard for Floating-Point Arithmetic (754-2008)”, *ISBN 978-0-7381-5753-5*, 2008
- [IeIp10] Institute of Electrical and Electronics Engineers, “IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows (1685-2009)”, *ISBN 978-0-7381-6160-0*, 2010
- [IeVe06] Institute of Electrical and Electronics Engineers, “IEEE Standard for Verilog Hardware Description Language (1364-2005)”, *ISBN 0-7381-4850-4*, 2006
- [IeVh09] Institute of Electrical and Electronics Engineers, “IEEE Standard VHDL Language Reference Manual (1076-2008)”, *ISBN 978-0-7381-5801-3*, 2009
- [Inte09] Intel Corp., “First the Tick, Now the Tock: Intel Microarchitecture (Nehalem)”, *White Paper No. 319724*, 2009
- [Inte99] International Business Machines Corp., “The CoreConnect Bus Architecture”, *White Paper*, 1999
- [IsoC10] International Organization for Standardization, “Road vehicles Controller area network (CAN)”, ISO 11898, 2003–2007
- [IsoE96] International Organization for Standardization, “ISO/IEC 14977:1996 Information technology – Syntactic metalanguage – Extended BNF”, 1996
- [IsoQ03] International Organization for Standardization, “ISO 10007:2003 Quality management systems - Guidelines for configuration management”, 2003
- [JaCh99] Jacob J. A., Chow P., “Memory Interfacing and Instruction Specification for Reconfigurable Processors”, Proc. 7th Intl. Symp. on Field Programmable Gate Arrays (FPGA), USA, 1999

- [JBPT02] Jones A., Bagchi D., Pal S., Tang X. et al., “PACT HDL: A C Compiler Targeting ASICs and FPGAs with Power and Performance Optimizations”, Proc. Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), France, 2002
- [JoBa02] Jones A., Banerjee P., “An Automated and Power-Aware Framework for Utilization of IP Cores in Hardware Generated from C Descriptions”, Technical Report: Center for Parallel and Distributed Computing, Northwestern University, *CPDC-TR-2002-04-002*, USA, 2002
- [John91] Johnson M., “Superscalar Microprocessor Design”, Prentice-Hall, 1991
- [KaKo05] Kasprzyk N., Koch A., “High-Level-Language Compilation for Reconfigurable Computers”, Proc. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC), France, 2005
- [KaKu07] Kachris C., Kulkarni C., “Configurable Transactional Memory”, Proc. 15th Symp. on Field-Programmable Custom Computing Machines (FCCM), USA, 2007
- [Kasp05] Kasprzyk, N., “COMRADE – Ein Hochsprachen-Compiler für Adaptive Computersysteme”, Ph.D. thesis, Tech. Univ. Braunschweig, 2005
- [KaYe05] Kaeli D. R., Yew P.-C. (Editors), “Speculative execution in high-performance computer architectures”, Chapman & Hall/CRC, 2005
- [KBWP07] Kelem S., Box B., Wasson S., Plunkett R. et al., “An Elemental Computing Architecture for SD Radio”, Proc. Software Defined Radio Technical Conf. and Product Exposition (SDR), USA, 2007
- [KCCM10] Koscher K., Czeskis A., Checkoway S., McCoy D. et al., “Experimental Security Analysis of a Modern Automobile”, IEEE Symposium on Security and Privacy, USA, 2010
- [KKGR03] Kasprzyk N., Koch A., Golze U., Rock M., “An Improved Intermediate Representation for Datapath Generation”, International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA, 2003
- [Koch00] Koch A., “Comprehensive Prototyping Platform for Hardware-Software Codesign”, Workshop on Rapid Systems Prototyping, Paris, France, 2000
- [Koch03] Koch A., “FLAME: A Flexible API for Module-based Environments - User’s Guide and Manual”, TU Braunschweig (E.I.S.), Braunschweig, Germany, 2003
- [Koch04] Koch. A., “Advances in Adaptive Computer Technology”, Habilitation thesis, Tech. Univ. Braunschweig, 2004

- [Koch07] Koch A., “Efficient Integration of Pipelined IP Blocks into Automatically Compiled Datapaths”, *EURASIP Journal on Embedded Systems*, Special Issue on Dynamically Reconfigurable Systems, 2007
- [Koch09] Koch A., “Adaptive Computing Systems and their Design Tools”, *in* *Dynamically Reconfigurable Systems* by Platzner M.; Teich J.; Wehn N. (Eds.), Springer, 2009
- [KWKS08] Kruijtzter W., Wolf P., Kock E., Stuyt J. et al., “Industrial IP Integration Flows based on IP-XACT Standards”, *Proc. Conf. on Design, Automation, and Test in Europe (DATE)*, Germany, 2008
- [LaKo00] Lange H., Koch A., “Memory Access Schemes for Configurable Processors”, *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, Villach, Austria, 2000
- [LaKo04] Lange H., Koch A., “Hardware/Software-Codesign by Automatic Embedding of Complex IP Cores”, *LNCS Intl. Conf. on Field-Programmable Logic (FPL)*, Antwerpen, 2004
- [LaKo07a] Lange H., Koch A., “Design and System Level Evaluation of a High Performance Memory System for reconfigurable SoC Platforms”, *Proc. HiPEAC Workshop on Reconfigurable Computing*, Ghent, 2007
- [LaKo07b] Lange H., Koch A., “An Execution Model for Hardware/Software Compilation and its System-Level Realization”, *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, Amsterdam, 2007
- [LaKo08] Lange H., Koch A., “Low-Latency High-Bandwidth HW/SW Communication in a Virtual Memory Environment”, *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, Heidelberg, 2008
- [LaKo10] Lange H., Koch A., “Architectures and Execution Models for Hardware/Software Compilation and their System-Level Realization”, *IEEE Transactions on Computers*, Vol. 59, No. 10, pp. 1363–1377, IEEE Computer Society Digital Library, 10/2010
- [Lam99] Lam M., “An Overview of the SUIF2 System”, *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, USA, 1999
- [Lang01] Lange H., “MARC: Ein parametrisiertes Speicherzugriffssystem für adaptive Rechner”, *Diploma Thesis*, TU Braunschweig (E.I.S.), Braunschweig, Germany, 2001
- [LaRa02] Lange H., Radetzki M., “IP Configuration Management with Abstract Parameterizations”, *Proc. International Workshop on IP Based SoC Design*, Grenoble, France, 2002

- [Laru99] Larus J., “Whole program paths”, Proc. ACM SIGPLAN ’99 Conference on Programming Language Design and Implementation, Atlanta, USA, 1999
- [Larz09] Larzul L., “EVE Taps Xilinx for Multiple Generations of Emulators”, Xcell Journal, Issue 68, Third Quarter 2009
- [Laur04] Laurich P., “A comparison of hard real-time Linux alternatives”, LinuxDevices, 2004
- [LaWK11] Lange H., Wink T., Koch A., “MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers”, Proc. Conf. on Design, Automation, and Test in Europe (DATE), France, 2011
- [LeFe09] Lewis B., Feiler P., “Architectural Computer System Model-Based Engineering with AADL”, Society of Automotive Engineers’ Architecture Analysis & Design Language Winter Meeting, Center for System and Software Engineering, University of Southern California, USA, 2009
- [LiRB01] Lin W. F., Reinhardt S. K., Burger D., “Reducing DRAM Latencies with an Integrated Memory Hierarchy Design”, Proc. 7th Intl. Symp. on High-Performance Computer Architecture (HPCA), Mexico, 2001
- [LSKH09] Lange H., Stock F., Koch A., Hildenbrand D., “Acceleration and Energy Efficiency of a Geometric Algebra Computation using Reconfigurable Computers and GPUs”, IEEE 17th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM), USA, 2009.
- [LSLB00] Lee M., Singh H., Lu G., Bagherzadeh N., Kurdahi F.J., “Design and Implementation of the MorphoSys Reconfigurable Computing Processor”, Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology, Kluwer, 2000
- [LuMG09] Lupon M., Magklis G., González A., “FasTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery”, Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT), USA, 2009
- [LuMo96] Luk C. K., Mowry T. C., “Compiler-Based Prefetching for Recursive Data Structures”, Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), USA, 1996
- [Macm01] MacMillen D., “Nimble Compiler Environment for Agile Hardware”, Storming Media LLC (USA), 2001
- [Magi10] Magillem S.A., “Magillem IP-XACT Packager”, Data Sheet, *Document Number: V.1.4 MIP Date April 2010*, 2010
- [MaGT98] Marcuello P., González A., Tubella J., “Speculative Multithreaded Processors”, Proc. 12th Intl. Conf. on Supercomputing, Australia, 1998

- [Mart07] Martynus M., “Branch Predictor Simulator”, Bachelor Thesis, TU Darmstadt, Germany, 2007
- [MBMH06] Moore K. E., Bobba J., Moravan M. J., Hill M. D., Wood D. A., “LogTM: Log-based Transactional Memory”, Intl. Symp. on High Performance Computer Architecture (HPCA), 2006
- [Ment10] Mentor Graphics Corp., “Catapult C Synthesis”, *Mentor MGC 01-10*, 2010
- [Merr10a] Merrit R., “Inside Intel’s Sandy Bridge architecture”, EE Times News, EE Times Group, 13/09/2010
- [Merr10b] Merrit R., “Intel rolls six merged Atom, FPGA chips”, EE Times News, EE Times Group, 22/11/2010
- [MTAB07] Majer M., Teich J., Ahmadiania A., Bobda C., “The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-Based Computer”, Journal of VLSI Signal Processing Systems, Springer, vol. 47(1), pp. 15–31, 03/2007
- [MüKo10] Mühlbach S., Koch A., “An FPGA-based Scalable Platform for High-Speed Malware Collection in Large IP Networks”, Proc. IEEE Intl. Conf. on Field-Programmable Technology (FPT), China, 2010
- [MüSc04] Müller-Hannemann M., Schnee M., “Finding All Attractive Train Connections by Multi-Criteria Pareto Search”, Proc. 4th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS), 11/2004
- [Nall10] Nallatech, “FPGA Accelerated Computing Solutions Intel Xeon Front Side Bus”, *Nallatech V1-5*, <http://www.nallatech.com>, 2010
- [NBDH05] Najjar W., Böhm W., Draper B., Hammes J. et al., “From Algorithms to Hardware – A High-Level Language Abstraction for Reconfigurable Computing”, IEEE Computer, 08/2005
- [NCVV03] Nollet V., Coene P., Verkest D., Vernalde S., Lauwereins R., “Designing an Operating System for a Heterogeneous Reconfigurable SoC”, Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), Nice, 2003
- [NCWT07] Njoroge N., Casper J., Wee S., Teslyar Y., Ge D., Kozyrakis C., Olukotun K., “ATLAS: A Chip-Multiprocessor with Transactional Memory Support”, Proc. Conf. on Design, Automation, and Test in Europe (DATE), France, 2007
- [NeDS04] Nesbit K. J., Dhodapkar A. S., Smith J. E., “AC/DC: An Adaptive Data Cache Prefetcher”, Proc. 13th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT), France, 2004

- [NeKo01] Neumann T., Koch. A., “A Generic Library for Adaptive Computing Environments”, Proc. Workshop on Field-Programmable Logic and Applications (FPL), Northern Ireland, 2001
- [NeSm05] Nesbit K. J., Smith J. E., “Data Cache Prefetching Using a Global History Buffer”, *Micro* 25(1), pp. 90–97, IEEE, 01/2005
- [Neum45] von Neumann J., “First Draft of a Report on the EDVAC”, Moore School of Electrical Engineering, University of Pennsylvania, USA, 1945
- [NVID06] NVIDIA Corp., “TurboCache Technology – Redefining Power/Performance and Price/Performance for GPU Solutions”, Technical Brief, *NVIDIA TB-01614-001_v03*, 2006
- [NXPS08] NXP Semiconductors, “SAA7160 PCI Express based audio and video bridge”, Product Data Sheet Rev. 01, *NXP SAA7160_1*, 2008
- [Oarc10] On-Line Applications Research Corp., “RTEMS Real-Time Executive for Multiprocessor Systems”, <http://www.rtems.com>, 2010
- [OBBG05] Ohmacht M., Bergamaschi R. A., Bhattacharya S., Gara A. et al, “Blue Gene/L compute chip: Memory and Ethernet subsystem”, IBM Journal of Research and Development, Vol. 49, No. 2/3, pp. 255–264, 2005
- [OhHW05] Ohlendorf R., Herkersdorf A., Wild T., “FlexPath NP: a network processor concept with application-driven flexible processing paths”, Proc. 3rd IEEE/ACM/IFIP Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS), USA, 2005
- [Omgi09] Object Management Group Inc., “UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems”, *OMG formal/2009-11-02*, 2009
- [Omgi10a] Object Management Group Inc., “OMG Unified Modeling Language (OMG UML), Infrastructure”, *OMG formal/2010-05-03*, 2010
- [Omgi10b] Object Management Group Inc., “OMG Unified Modeling Language (OMG UML), Superstructure”, *OMG formal/2010-05-05*, 2010
- [Open09] Open Core Protocol International Partnership Association Inc., “Open Core Protocol (OCP) Specification 3.0”, <http://www.ocpip.org>, 2009
- [PBDM08] Putnam A., Bennett D., Dellinger E., Mason J., Sundararajan P., Eggers S., “CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures”, Proc. Int. Conference on Field Programmable Logic and Applications (FPL), 2008
- [Peri10] Peripheral Component Interconnect Special Interest Group (PCI-SIG), “PCI Express Base Specification 2.1”, <http://www.pcisig.com/specifications/pciexpress>, 2010

- [Phil04] Philips Semiconductors, “SAA7146A Multimedia bridge, high performance Scaler and PCI circuit (SPCI)”, Product Specification, 2004
- [PLXT98] PLX Technology Inc., “PCI 9080 I₂O Compatible PCI Bus Master I/O Accelerator Chip”, *PLX 9080-PB-010*, 1998
- [PoCT09] Porter L., Choi B., Tullsen D. M., “Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading”, Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT), USA, 2009
- [Pose06a] Poseidon Design Systems Inc., “Triton Builder – Xilinx FPGA Systems v4.3”, http://www.poseidon-systems.com/builder_xilinx.pdf, retrieved 10/2010, 2006
- [Pose06b] Poseidon Design Systems Inc., “Triton tools and Xilinx EDK Design Flow”, http://www.poseidon-systems.com/interface_xilinx.pdf, retrieved 10/2010, 2006
- [RMMT10] Rouf I., Miller R., Mustafa H., Taylor T. et al., “Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study”, Proc. 19th USENIX Security Symposium, Washington, USA, 2010
- [Schu09] Schuler E., “NoC Concepts with XPP-III”, ESTEC Network on Chip round table, European Space Agency, The Netherlands, 2009
- [ScJA99] Schubert A., Jährg R., Anheier W., “Cryptographic Reuse Library”, Proc. Forum on Design Languages (FDL), France, 1999
- [ScSa07] Schmidt, A. G., Sass, R., “Quantifying Effective Memory Bandwidth of Platform FPGAs”, Proc. 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, 2007
- [SDBM03] Sethumadhavan S., Desikan R., Burger D., Moore C. R., Keckler S. W., “Scalable Hardware Memory Disambiguation for High ILP Processors”, 36th Intl. Symp. on Microarchitecture (MICRO-36), USA, 2003
- [SeLK02] Sendag R., Lilja D. J., Kunkel S. R., “Exploiting the Prefetching Effect Provided by Executing Mispredicted Load Instructions”, ACM Euro-Par Conference, Germany, 2002
- [SGNB08] Slognat D., Giese A., Nüssle M., Brüning U., “An Open-Source Hypertransport Core”, ACM Trans. on Reconfigurable Technology and Systems, Vol. 1, No. 3, pp. 1–21, 2008
- [ShLi05] Shen J. P., Lipasti M., “Modern Processor Design”, McGraw-Hill, 2005

- [Sili03] Silicon Hive, “Reconfigurable accelerators that bring computational efficiency (MOPS/W) and programmability together, to displace ASIC and DSP co-processors in Systems-on-Chips”, Silicon Hive Technology Primer, 2003
- [SLLM06] Schweitz E., Lethin R., Leung A., Meister B., “R-Stream: A Parametric High Level Compiler”, Proc. 10th Annual Workshop on High Performance Embedded Computing (HPEC), USA, 2006
- [SoTB06] So H., Tkachenko A., Brodersen R., “A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH”, Proc. 16th Int. Conf. on Field Programmable Logic and Applications (FPL), Madrid, 2006
- [SPEC06] SPEC CPU Subcommittee, original program authors, “SPEC CPU2006 Benchmark Descriptions”, SPEC - Standard Performance Evaluation Corp., 2006
- [Stal02] Stallman R. M., “Using the GNU Compiler Collection for GCC 3.3.6”, <http://gcc.gnu.org/onlinedocs>, retrieved 10/2010, 2002
- [StKA99] Strøm Ø., Klauseie A., Aas E. J., “A Study of Dynamic Instruction Frequencies in Byte Compiled Java Programs”, Proc. 25th Euromicro Conference, Italy, 1999
- [Stre09] Stretch Inc., “S6000 Family Product Brief”, *Stretch MK-6000C-0001-001*, 2009
- [Stre10] Stretch Inc., “S7000 Family Product Brief”, *Stretch MK-7000-0001-000*, 2010
- [StWF05] Stone S. S., Woley K. M., Frank M. I., “Address-Indexed Memory Disambiguation and Store-to-Load Forwarding”, Proc. Intl. Symp. on Microarchitecture (MICRO-38), Spain, 2005
- [Sunm97] Sun Microsystems Inc., “microSPARC-IIep User’s Manual”, *Part Number: 802-7100-01*, April 1997
- [SvGB05] Svahnberg M., van Gorp J., Bosch J., “A Taxonomy of Variability Realization Techniques”, *Software: Practice and Experience*, Vol. 35(8), pp. 1–50, Wiley, 2005
- [SWAF06] Somogyi S., Wenisch T. F., Ailamaki A., Falsafi B., Moshovos A., “Spatial Memory Streaming”, *SIGARCH Computer Architecture News* 34(2), pp. 252–263, 2006
- [SWMH09] Shoufan A., Wink T., Molter H. G., Huss S. A., Strenzke F., “A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms”, IEEE 20th Intl. Conf. on Application-specific Systems, Architectures and Processors (ASAP), USA, 2009

- [Syno08] Synopsys Inc., “Synopsys coreTools – IP Based Design and Verification”, Datasheet, *Synopsys 06/08.PS.WO.08-16501*, 2008
- [Syno10a] Synopsys Inc., “Synphony C Compiler – Optimized Hardware from High-Level C/C++”, Datasheet, *Synopsys 10/10.TT.10-19109*, 2010
- [Syno10b] Synopsys Inc., “Synphony Model Compiler – High-Level Synthesis for Model-Based Design”, Datasheet, *Synopsys 08/10.CE.10-18924*, 2010
- [Synp08] Synplicity Inc., “Synplify DSP ESL Synthesis Datasheet”, *Synopsys 20708DSP*, 2008
- [Taiw05] Taiwan Semiconductor Manufacturing Company Ltd., “TSMC 0.18 and 0.15-micron Technology Platform”, *TSMC 2000-5/04.05*, 2005
- [TGHD99] Tomiyama H., Grun P., Halambi A., Dutt N., and Nicolau A., “Architecture Description Languages for Systems-on-Chip Design”, 6th Asia Pacific Conference on Chip Design Language, Fukuoka, Japan, 1999
- [ThHa08] Thatcher T., Hartke P., “OpenSPARC T1 on Xilinx FPGAs - Updates”, RAMP Retreat, Stanford, 08/2008
- [Thro00] Thronicke W., “Konzept und Realisierung einer allgemeinen Parametrisierungsstrategie von Systemmodellen unter besonderer Berücksichtigung der Wiederverwendbarkeit”, Ph.D. thesis, Universität Paderborn, Germany, 2000
- [TISC95] TIS Committee, Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, 1995
- [Tosh10] Toshiba Corp., “Toshiba Unveils CELL REGZA LCD TV Series with Superior 3D Capabilities”, News Release, July 28th, 2010
- [Tris02] Triscend Inc., “Triscend a7s configurable system-on-chip platform”, Data Sheet 1.10, 2002
- [Turl06] Turley J., “Operating Systems on the Rise”, <http://www.embedded.com/columns/showArticle.jhtml?articleID=187203732>, retrieved 10/2010, 2006
- [VDCR08] VDC Research, “2008 Embedded Software Market Intelligence”, *white paper*, 2008
- [VoRH09] Voros N., Rosti A., Hübner M. (Eds.), “Dynamic System Reconfiguration in Heterogeneous Platforms – The MORPHEUS Approach”, Springer, 2009
- [VuPI04] Vuletić M., Pozzi L., Ienne P., “Virtual Memory Window for Application-Specific Reconfigurable Coprocessors”, Proc. Design Automation Conference (DAC), San Diego, 2004

- [VuPI05] Vuletić M., Pozzi L., Ienne P., “Seamless hardware-software integration in reconfigurable computing systems”, Design & Test of Computers, Vol 22 , Issue 2, pp. 102–113, IEEE, 03/2005
- [WaBL06] Wang X., Braganza S., Leiser M., “Advanced Components in the Variable Precision Floating-Point Library”, Proc. 14th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), USA, 2006
- [Wall91] Wall D., “Limits of instruction-level parallelism”, Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), USA, 1991
- [Walr10] Walrath J., “AMD’s Ontario and Zacate APUs Further Exposed”, PC Perspective, <http://www.pcper.com/article.php?aid=996>, 09/09/2010
- [Will10] Williston K., “Intel QPI accelerator sneak preview”, Hardware Blog, Intel Embedded Community, <http://community.edc.intel.com>, 04-06-2010
- [Wind10] Wind River Systems Inc., “The Crisis of Complexity”, http://www.windriver.com/products/test_management/survey-0610.pdf, retrieved 09/2010, 2010
- [WZGB00] Wan M., Zhang H., George V., Benes M., Abnous A., Prabhu V., Rabaey J., “Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System”, Journal of VLSI Signal Processing, 2000
- [Xili01] Xilinx Inc., “High-Performance 16-Point Complex FFT/IFFT V1.0 Product Specification”, 2001
- [Xili05a] Xilinx Inc., “ML310 User Guide”, *Xilinx UG068*, 2005
- [Xili05b] Xilinx Inc., “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet”, *Xilinx DS083*, 2005
- [Xili05c] Xilinx Inc., “LocalLink Interface Specification”, *Xilinx SP006 (v2.0)*, 2005
- [Xili06] Xilinx Inc., “Embedded System Tools Reference Manual”, *Xilinx UG111*, 2006
- [Xili09a] Xilinx Inc., “Virtex-5 FXT PowerPC 440 and MicroBlaze Edition Kit Reference Systems”, *Xilinx UG511*, 2009
- [Xili09b] Xilinx Inc., “System Generator for DSP User Guide”, *Xilinx UG640*, 2009
- [Xili09c] Xilinx Inc., “ML505/ML506/ML507 Evaluation Platform User Guide”, *Xilinx UG347*, 2009
- [Xili10a] Xilinx Inc., “Virtex-5 FPGA User Guide”, *Xilinx UG190*, 2010

- [Xili10b] Xilinx Inc., “MicroBlaze Processor Reference Guide”, *Xilinx UG081*, 2010
- [Xili10c] Xilinx Inc., “PicoBlaze 8-bit Embedded Microcontroller User Guide”, *Xilinx UG129*, 2010
- [Xili10d] Xilinx Inc., “OS and Libraries Document Collection”, *Xilinx UG 643*, 2010
- [Xili10e] Xilinx Inc., “7 Series FPGAs Overview”, *Xilinx DS180 (v1.2)*, 2010
- [Xili10f] Xilinx Inc., “Embedded Processor Block in Virtex-5 FPGAs Reference Guide”, *Xilinx UG200 (v1.8)*, 2010
- [Xili10g] Xilinx Inc., “IP Documentation”, http://www.xilinx.com/support/documentation/ip_documentation.htm, retrieved 10/2010, 2010
- [YBMM07] Yen L., Bobba J., Marty M. R., Moore K. E. et al., “LogTM-SE: Decoupling Hardware Transactional Memory from Caches”, Proc. 13th Intl. Symp. on High Performance Computer Architecture (HPCA), USA, 2007
- [YePa92] Yeh T. Y., Patt Y. N., “Alternative Implementations of Two-Level Adaptive Branch Prediction”, Proc. 19th Intl. Symp. on Computer Architecture (ISCA), Australia, 1992
- [Zell97] Zeller A., “Configuration Management with Version Sets”, Ph.D. thesis, Technische Universität Braunschweig, Germany, 1997
- [ZyVS08] Zyś M., Vaumorin E., Sobański I., “Straightforward IP Integration with IP-XACT RTL-TLM Switching”, Design Automation Conference (DAC), USA, 2008

A. Appendix

Listing A.1: CoMAP interface templates for Xilinx ML507

```
comap 1.0
; interface templates
interface plb_clk_reset
  type: plb
  version: 4.6
  comment
    "PLB clock and reset generator"
  end comment
  port clk
    transaction: clock min: 100 MHz max: 100 MHz
    direction: output
    width: 1
    access: master
  end port
  port reset
    transaction: posreset 100 cycles
    direction: output
    width: 1
    access: master
  end port
end interface

interface PLB_master
  type: plb
  version: 4.6
  comment
    "PLB master template"
  end comment
  port clk
    transaction: clock min: 100 MHz max: 100 MHz
    direction: input
    width: 1
    access: slave
  end port
  port reset
```

A. Appendix

```
    transaction: posreset 100 cycles
    direction: input
    width: 1
    access: slave
end port
port request
    comment
        "Master request to arbiter"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
    enablein name: addrack offset: 0 latency: 4
end port
port priority
    comment
        "Master priority to arbiter"
    end comment
    transaction: data
    direction: output
    width: 2
    clock: clk polarity: pos
    enableout name: request offset: 0 latency: 4
    enablein name: addrack offset: 0 latency: 4
end port
port read_not_write
    comment
        "Transfer direction"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
    enableout name: request offset: 0 latency: 4
    enablein name: addrack offset: 0 latency: 4
    access: master priority: priority request: request grant: addrack
end port
port addr
    comment
        "Master address"
    end comment
    transaction: data
    direction: output
```



```

width: param_plb_addr_width
clock: clk polarity: pos
enableout name: request offset: 0 latency: 4
enablein name: addrack offset: 0 latency: 4
access: master priority: priority request: request grant: addrack
end port
port be
comment
    "Byte enable"
end comment
transaction: data
direction: output
width: param_plb_be_width
clock: clk polarity: pos
enableout name: request offset: 0 latency: 4
enablein name: addrack offset: 0 latency: 4
access: master priority: priority request: request grant: addrack
end port
port writedata
comment
    "Master write bus"
end comment
transaction: data
direction: output
width: param_plb_data_width
clock: clk polarity: pos
address port: addr
    physical: param_plb_addr_width-1:0
    logical: param_plb_addr_width-1:0
enablein name: writeack offset: 0 latency: 4
access: master priority: priority request: request grant: addrack
end port
port readdata
comment
    "Master read bus"
end comment
transaction: data
direction: input
width: param_plb_data_width
clock: clk polarity: pos
address port: addr
    physical: param_plb_addr_width-1:0
    logical: param_plb_addr_width-1:0
enablein name: readack offset: 0 latency: 4

```

A. Appendix

```
    access: master priority: priority request: request grant: addrack
end port
port addrack
    comment
        "Address acknowledge"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enableout name: request offset: 0 latency: 4
    access: master priority: priority request: request grant: addrack
end port
port readack
    comment
        "Read acknowledge"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    access: master priority: priority request: request grant: addrack
end port
port writeack
    comment
        "Write acknowledge"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    access: master priority: priority request: request grant: addrack
end port
end interface

interface PLB_slave
    type: plb
    version: 4.6
    comment
        "PLB slave template"
    end comment
    port clk
        transaction: clock min: 100 MHz max: 100 MHz
        direction: input
```

```

    width: 1
    access: slave
end port
port reset
    transaction: posreset 100 cycles
    direction: input
    width: 1
    access: slave
end port
port request
    comment
        "Arbiter request to slave"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enableout name: addrack offset: 0 latency: 4
    access: slave
end port
port read_not_write
    comment
        "Transfer direction"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enablein name: request offset: 0 latency: 4
    enableout name: addrack offset: 0 latency: 4
    access: slave
end port
port addr
    comment
        "Address"
    end comment
    transaction: data
    direction: input
    width: param_plb_addr_width
    clock: clk polarity: pos
    enablein name: request offset: 0 latency: 4
    enableout name: addrack offset: 0 latency: 4
    access: slave
end port

```

A. Appendix

```
port be
  comment
    "Byte enable"
  end comment
  transaction: data
  direction: input
  width: param_plb_be_width
  clock: clk polarity: pos
  enablein name: request offset: 0 latency: 4
  enableout name: addrack offset: 0 latency: 4
  access: slave
end port
port writedata
  comment
    "Write bus"
  end comment
  transaction: data
  direction: input
  width: param_plb_data_width
  clock: clk polarity: pos
  address port: addr
    physical: param_plb_addr_width-1:0
    logical: param_plb_addr_width-1:0
  enableout name: writeack offset: 0 latency: 4
  access: slave
end port
port readdata
  comment
    "Read bus"
  end comment
  transaction: data
  direction: output
  width: param_plb_data_width
  clock: clk polarity: pos
  address port: addr
    physical: param_plb_addr_width-1:0
    logical: param_plb_addr_width-1:0
  enablein name: readack offset: 0 latency: 4
  access: slave
end port
port addrack
  comment
    "Address acknowledge"
  end comment
```

```

    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
    enablein name: request offset: 0 latency: 4
    access: slave
end port
port readack
    comment
        "Read acknowledge"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
    access: slave
end port
port writeack
    comment
        "Write acknowledge"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
    access: slave
end port
end interface

interface LocalLink_source
    type: LocalLink
    version: 1.0
    comment
        "Xilinx LocalLink DMA source"
    end comment
    port clk
        transaction: clock min: 100 MHz max: 125 MHz
        direction: output
        width: 1
    end port
    port reset
        transaction: posreset 100 cycles
        direction: output
        width: 1

```

A. Appendix

```
end port
port data
  transaction: data
  direction: output
  width: 32
  clock: clk polarity: pos
  ; Header
  sequence repeat: 8
    bigendian: 32 bit unsigned
  end sequence
  ; Payload
  ; length signalled by start_of_payload/end_of_payload
  sequence repeat: inf
    bigendian: 32 bit unsigned
  end sequence
  ; Footer
  sequence
    bigendian: 32 bit unsigned
  end sequence
  enableout name: source_ready offset: 0 latency: 0
  enablein name: destination_ready offset: 0 latency: 0
end port
port source_ready
  comment
    "Source handshake"
  end comment
  transaction: data
  direction: output
  width: 1
  clock: clk polarity: pos
end port
port destination_ready
  comment
    "Destination handshake"
  end comment
  transaction: data
  direction: input
  width: 1
  clock: clk polarity: pos
end port
port remain
  comment
    "Valid bits of last byte"
  end comment
```

```

transaction: data
direction: output
width: 4
clock: clk polarity: pos
enableout name: source_ready offset: 0 latency: 0
enablein name: destination_ready offset: 0 latency: 0
end port
port start_of_frame
comment
    "Marks start of header transmission"
end comment
transaction: data
direction: output
width: 1
clock: clk polarity: pos
enableout name: source_ready offset: 0 latency: 0
enablein name: destination_ready offset: 0 latency: 0
end port
port start_of_payload
comment
    "Marks start of payload transmission"
end comment
transaction: data
direction: output
width: 1
clock: clk polarity: pos
enableout name: source_ready offset: 0 latency: 0
enablein name: destination_ready offset: 0 latency: 0
end port
port end_of_payload
comment
    "Marks end of payload transmission"
end comment
transaction: data
direction: output
width: 1
clock: clk polarity: pos
enableout name: source_ready offset: 0 latency: 0
enablein name: destination_ready offset: 0 latency: 0
end port
port end_of_frame
comment
    "Marks start of footer transmission"
end comment

```

A. Appendix

```
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
    enableout name: source_ready offset: 0 latency: 0
    enablein name: destination_ready offset: 0 latency: 0
end port
end interface

interface LocalLink_destination
    type: LocalLink
    version: 1.0
    comment
        "Xilinx LocalLink DMA destination"
    end comment
    port clk
        transaction: clock min: 100 MHz max: 125 MHz
        direction: input
        width: 1
    end port
    port reset
        transaction: posreset 100 cycles
        direction: input
        width: 1
    end port
    port data
        transaction: data
        direction: input
        width: 32
        clock: clk polarity: pos
        ; Header
        sequence repeat: 8
            bigendian: 32 bit unsigned
        end sequence
        ; Payload
        ; length signalled by start_of_payload/end_of_payload
        sequence repeat: inf
            bigendian: 32 bit unsigned
        end sequence
        ; Footer
        sequence
            bigendian: 32 bit unsigned
        end sequence
        enablein name: source_ready offset: 0 latency: 0
```



```

    enableout name: destination_ready offset: 0 latency: 0
end port
port source_ready
    comment
        "Source handshake"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
end port
port destination_ready
    comment
        "Destination handshake"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
end port
port remain
    comment
        "Valid bits of last byte"
    end comment
    transaction: data
    direction: input
    width: 4
    clock: clk polarity: pos
    enablein name: source_ready offset: 0 latency: 0
    enableout name: destination_ready offset: 0 latency: 0
end port
port start_of_frame
    comment
        "Marks start of header transmission"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enablein name: source_ready offset: 0 latency: 0
    enableout name: destination_ready offset: 0 latency: 0
end port
port start_of_payload
    comment

```

A. Appendix

```
    "Marks start of payload transmission"
end comment
transaction: data
direction: input
width: 1
clock: clk polarity: pos
enablein name: source_ready offset: 0 latency: 0
enableout name: destination_ready offset: 0 latency: 0
end port
port end_of_payload
    comment
        "Marks end of payload transmission"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enablein name: source_ready offset: 0 latency: 0
    enableout name: destination_ready offset: 0 latency: 0
end port
port end_of_frame
    comment
        "Marks start of footer transmission"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enablein name: source_ready offset: 0 latency: 0
    enableout name: destination_ready offset: 0 latency: 0
end port
end interface
end comap
```

Listing A.2: CoMAP component library for Xilinx ML507

```
comap 1.0
; component library
component PLB_CLK_RESET
  version: 1.0
  comment
    "PLB clock and reset generator"
  end comment
  technology: "XC5VFX70T-1"
  speed min: 0 MHz max: 100 MHz

  interface clk_reset
    template: plb_clk_reset
  end interface
end component

component Interrupt_Controller
  version: 1.0
  comment
    "Interrupt controller"
  end comment
  technology: "XC5VFX70T-1"
  speed min: 0 MHz max: 100 MHz

  interface clk_reset
    type: plb
    version: 4.6
    comment
      "PLB clock and reset"
    end comment
    port clk
      transaction: clock min: 100 MHz max: 100 MHz
      direction: input
      width: 1
      access: slave
    end port
    port reset
      transaction: posreset 100 cycles
      direction: input
      width: 1
      access: slave
    end port
  end interface
end component
```

A. Appendix

```
interface PLB
  template: PLB_slave
end interface

interface interrupt_cpu
  type: irq_cpu
  version: 1.0
  comment
    "Forward IRQ to CPU"
  end comment
  port irq
    transaction: interrupt
    direction: output
    width: 1
  end port
end interface

interface interrupt_slave
  type: irq_slave
  version: 1.0
  comment
    "IRQ from slave"
  end comment
  port irq_ethernet
    transaction: interrupt
    direction: input
    width: 1
  end port
  port irq_uart
    transaction: interrupt
    direction: input
    width: 1
  end port
end interface
end component

component PowerPC440
  version: 1.0
  comment
    "PowerPC 440 processor block"
  end comment
  technology: "XC5VFX70T-1"
  speed min: 0 MHz max: 400 MHz
```

```

interface PLB
    template: PLB_master
end interface

interface LocalLink_write
    template: LocalLink_source
end interface

interface LocalLink_read
    template: LocalLink_destination
    comment
        "Frame read protocol"
    end comment
    port data
        transaction: data
        direction: input
        width: 32
        clock: clk polarity: pos
        ; Header
        sequence
            bigendian: 32 bit unsigned
        end sequence
        ; Payload
        ; length signalled by start_of_payload/end_of_payload
        sequence repeat: inf
            bigendian: 32 bit unsigned
        end sequence
        ; Footer
        sequence repeat: 8
            bigendian: 32 bit unsigned
        end sequence
        enablein name: source_ready offset: 0 latency: 0
        enableout name: destination_ready offset: 0 latency: 0
    end port
end interface

interface MCI_master
    exclusive
    type: MCI
    version: 1.0
    comment
        "Xilinx Memory Controller Interface master"
    end comment

```

A. Appendix

```
port clk
  transaction: clock min: 200 MHz max: 200 MHz
  direction: output
  width: 1
end port
port reset
  transaction: posreset 100 cycles
  direction: output
  width: 1
end port
port read_not_write
  comment
    "Transfer direction"
  end comment
  transaction: data
  direction: output
  width: 1
  clock: clk polarity: pos
  enablein name: address_ready_to_accept offset: -2 latency: 0
  enableout name: address_valid offset: 0 latency: 0
end port
port address
  transaction: data
  direction: output
  width: 36
  clock: clk polarity: pos
  enablein name: address_ready_to_accept offset: -2 latency: 0
  enableout name: address_valid offset: 0 latency: 0
end port
port address_valid
  comment
    "Address handshake"
  end comment
  transaction: data
  direction: output
  width: 1
  clock: clk polarity: pos
  enablein name: address_ready_to_accept offset: -2 latency: 0
end port
port write_data
  transaction: data
  direction: output
  width: 128
  clock: clk polarity: pos
```

```

    address port: address
      physical: 35:0
      logical: 35:0
      enableout name: write_data_valid offset: 0 latency: 0
    end port
port byte_enable
  comment
    "Write byte lanes"
  end comment
  transaction: data
  direction: output
  width: 16
  clock: clk polarity: pos
  enableout name: write_data_valid offset: 0 latency: 0
end port
port write_data_valid
  comment
    "Write data handshake"
  end comment
  transaction: data
  direction: output
  width: 1
  clock: clk polarity: pos
end port
port bank_conflict
  comment
    "New bank address"
  end comment
  transaction: data
  direction: output
  width: 1
  clock: clk polarity: pos
  enablein name: address_ready_to_accept offset: -2 latency: 0
  enableout name: address_valid offset: 0 latency: 0
end port
port row_conflict
  comment
    "New row address"
  end comment
  transaction: data
  direction: output
  width: 1
  clock: clk polarity: pos
  enablein name: address_ready_to_accept offset: -2 latency: 0

```

A. Appendix

```
    enableout name: address_valid offset: 0 latency: 0
end port
port read_data
    transaction: data
    direction: input
    width: 128
    clock: clk polarity: pos
    address port: address
        physical: 35:0
        logical: 35:0
    enablein name: read_data_valid offset: 0 latency: 0
end port
port read_data_valid
    comment
        "Read data handshake"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
end port
port read_data_err
    comment
        "Read data ECC failed"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enablein name: read_data_valid offset: 0 latency: 0
end port
port address_ready_to_accept
    comment
        "Memory controller ready"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
end port
end interface

interface interrupt
    type: irq_cpu
```



```

version: 1.0
comment
    "IRQ input from interrupt controller"
end comment
port irq
    comment
        "External interrupt request"
    end comment
    transaction: interrupt
    direction: input
    width: 1
end port
end interface

```

```

parameter param_plb_addr_width
    location: "ML507/PPC440"
    value: "32"
    type: integer
    modifiable: no
    strippable: yes
end parameter

```

```

parameter param_plb_data_width
    location: "ML507/PPC440"
    value: "64"
    type: integer
    modifiable: no
    strippable: yes
    dependants
        name: param_plb_be_width
        location: "ML507/PPC440"
    end dependants
end parameter

```

```

parameter param_plb_be_width
    location: "ML507/PPC440"
    value: "8"
    type: integer
    modifiable: yes
    strippable: yes
    boundary
        is: param_plb_data_width / 4
    end boundary
end parameter

```

A. Appendix

end component

component *Ethernet*

version: 1.01

comment

"Ethernet 1G NIC"

end comment

technology: *"XC5VFX70T-1"*

area

luts min: 748 max: 7085

flipflops min: 920 max: 5234

brams min: 6 max: 76

end area

speed min: 100 MHz max: 100 MHz

interface *PLB*

template: *PLB_slave*

end interface

interface *LocalLink_write*

template: *LocalLink_destination*

end interface

interface *LocalLink_read*

template: *LocalLink_source*

comment

"Frame read protocol"

end comment

port data

transaction: data

direction: output

width: 32

clock: *clk* polarity: *pos*

; Header

sequence

bigendian: 32 bit unsigned

end sequence

; Payload

; length signalled by start_of_payload/end_of_payload

sequence repeat: inf

bigendian: 32 bit unsigned

end sequence

; Footer

sequence repeat: 8

```

        bigendian: 32 bit unsigned
    end sequence
    enableout name: source_ready offset: 0 latency: 0
    enablein name: destination_ready offset: 0 latency: 0
end port
end interface

interface interrupt
    type: irq_slave
    version: 1.0
    comment
        "IRQ request to interrupt controller"
    end comment
    port irq_ethernet
        comment
            "Slave interrupt request"
        end comment
        transaction: interrupt
        direction: output
        width: 1
    end port
end interface

parameter param_plb_addr_width
    location: "ML507/Ethernet"
    value: "32"
    type: integer
    modifiable: no
    strippable: yes
end parameter

parameter param_plb_data_width
    location: "ML507/Ethernet"
    value: "64"
    type: integer
    modifiable: no
    strippable: yes
    dependants
        name: param_plb_be_width
        location: "ML507/Ethernet"
    end dependants
end parameter

parameter param_plb_be_width

```

A. Appendix

```
location: "ML507/Ethernet"
value: "8"
type: integer
modifiable: yes
strippable: yes
boundary
    is: param_plb_data_width / 4
end boundary
end parameter
end component

component UART
version: 1.0
comment
    "RS232 interface"
end comment
technology: "XC5VFX70T-1"
area
    luts min: 292 max: 409
    flipflops min: 258 max: 379
end area
speed min: 100 MHz max: 215 MHz

interface PLB
    template: PLB_slave
end interface

interface interrupt
    type: irq_slave
    version: 1.0
    comment
        "IRQ request to interrupt controller"
    end comment
    port irq_uart
        comment
            "Slave interrupt request"
        end comment
        transaction: interrupt
        direction: output
        width: 1
    end port
end interface

parameter param_plb_addr_width
```

```

    location: "ML507/UART"
    value: "32"
    type: integer
    modifiable: no
    strippable: yes
end parameter

parameter param_plb_data_width
    location: "ML507/UART"
    value: "64"
    type: integer
    modifiable: no
    strippable: yes
    dependants
        name: param_plb_be_width
        location: "ML507/UART"
    end dependants
end parameter

parameter param_plb_be_width
    location: "ML507/UART"
    value: "8"
    type: integer
    modifiable: yes
    strippable: yes
    boundary
        is: param_plb_data_width / 4
    end boundary
end parameter
end component

component DDR2_Controller
    version: 2.0
    comment
        "DDR2 SDRAM controller"
    end comment
    technology: "XC5VFX70T-1"
    speed min: 200 MHz max: 200 MHz

    interface MCI_slave
        resource
        type: MCI
        version: 1.0
        comment

```

A. Appendix

```
"Xilinx Memory Controller Interface slave"
end comment
port clk
  transaction: clock min: 200 MHz max: 200 MHz
  direction: input
  width: 1
end port
port reset
  transaction: posreset 100 cycles
  direction: input
  width: 1
end port
port read_not_write
  comment
    "Transfer direction"
  end comment
  transaction: data
  direction: input
  width: 1
  clock: clk polarity: pos
  enableout name: address_ready_to_accept offset: -2 latency: 0
  enablein name: address_valid offset: 0 latency: 0
end port
port address
  transaction: data
  direction: input
  width: 36
  clock: clk polarity: pos
  enableout name: address_ready_to_accept offset: -2 latency: 0
  enablein name: address_valid offset: 0 latency: 0
end port
port address_valid
  comment
    "Address handshake"
  end comment
  transaction: data
  direction: input
  width: 1
  clock: clk polarity: pos
  enableout name: address_ready_to_accept offset: -2 latency: 0
end port
port write_data
  transaction: data
  direction: input
```

```

width: 128
clock: clk polarity: pos
address port: address
    physical: 35:0
    logical: 35:0
    enablein name: write_data_valid offset: 0 latency: 0
end port
port byte_enable
    comment
        "Write byte lanes"
    end comment
    transaction: data
    direction: input
    width: 16
    clock: clk polarity: pos
    enablein name: write_data_valid offset: 0 latency: 0
end port
port write_data_valid
    comment
        "Write data handshake"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
end port
port bank_conflict
    comment
        "New bank address"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enableout name: address_ready_to_accept offset: -2 latency: 0
    enablein name: address_valid offset: 0 latency: 0
end port
port row_conflict
    comment
        "New row address"
    end comment
    transaction: data
    direction: input
    width: 1

```

A. Appendix

```
    clock: clk polarity: pos
    enableout name: address_ready_to_accept offset: -2 latency: 0
    enablein name: address_valid offset: 0 latency: 0
end port
port read_data
    transaction: data
    direction: output
    width: 128
    clock: clk polarity: pos
    address port: address
        physical: 35:0
        logical: 35:0
    enableout name: read_data_valid offset: 0 latency: 0
end port
port read_data_valid
    comment
        "Read data handshake"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
end port
port read_data_err
    comment
        "Read data ECC failed"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
    enableout name: read_data_valid offset: 0 latency: 0
end port
port address_ready_to_accept
    comment
        "Memory controller ready"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
end port
end interface
```



```

interface FastLane
  resource
  type: FastLane
  version: 2.0
  comment
    "FastLane+ hardware accelerator docking point"
  end comment
  port clk
    transaction: clock min: 100 MHz max: 200 MHz
    direction: output
    width: 1
  end port
  port reset
    transaction: posreset 100 cycles
    direction: output
    width: 1
  end port
  ; Slave side ports (from CPU)
  port addr
    comment
      "Address from CPU"
    end comment
    transaction: data
    direction: output
    width: 32
    clock: clk polarity: pos
    enableout name: addressed offset: 0 latency: 0
  end port
  port addressed
    comment
      "CPU address is valid"
    end comment
    transaction: data
    direction: output
    width: 1
    clock: clk polarity: pos
  end port
  port write
    comment
      "This is a write, not read"
    end comment
    transaction: data
    direction: output
    width: 1

```

A. Appendix

```
    clock: clk polarity: pos
    enableout name: addressed offset: 0 latency: 0
end port
port data_in
    comment
        "Write data from CPU"
    end comment
    transaction: data
    direction: output
    width: 32
    clock: clk polarity: pos
    address port: addr
        physical: 31:0
        logical: 31:0
    enableout name: addressed offset: 0 latency: 0
end port
port data_out
    comment
        "Read data to CPU"
    end comment
    transaction: data
    direction: input
    width: 32
    clock: clk polarity: pos
    address port: addr
        physical: 31:0
        logical: 31:0
    enableout name: addressed offset: -1 latency: 0
end port
; Master side ports (to DDR2 RAM)
port output_enable
    comment
        "Address valid, read data"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    stallout name: stall offset: 0 latency: 4
end port
port byte_write_enable
    comment
        "Address valid, write data bytes"
    end comment
```

```

        transaction: data
        direction: input
        width: 16
        clock: clk polarity: pos
        stallout name: stall offset: 0 latency: 4
    end port
    port ram_address
        transaction: data
        direction: input
        width: 32
        clock: clk polarity: pos
        stallout name: stall offset: 0 latency: 4
    end port
    port write_data
        transaction: data
        direction: input
        width: 128
        clock: clk polarity: pos
        address port: ram_address
            physical: 31:0
            logical: 31:0
        stallout name: stall offset: 0 latency: 4
    end port
    port read_data
        transaction: data
        direction: output
        width: 128
        clock: clk polarity: pos
        address port: ram_address
            physical: 31:0
            logical: 31:0
        stallout name: stall offset: -1 latency: 4
    end port
    port stall
        comment
            "FastLane+ is busy, wait"
        end comment
        transaction: data
        direction: output
        width: 1
        clock: clk polarity: pos
    end port
end interface
end component

```

A. Appendix

```
component HA
  version: 1.14
  comment
    "Hardware accelerator"
  end comment
  technology: "XC5VFX70T-1"
  speed min: 100 MHz max: 168 MHz

interface FastLane
  exclusive
  type: FastLane
  version: 2.0
  comment
    "FastLane+ hardware accelerator user logic"
  end comment
  port clk
    transaction: clock min: 100 MHz max: 200 MHz
    direction: input
    width: 1
  end port
  port reset
    transaction: posreset 100 cycles
    direction: input
    width: 1
  end port
  ; Slave side ports (from CPU)
  port addr
    comment
      "Address from CPU"
    end comment
    transaction: data
    direction: input
    width: 32
    clock: clk polarity: pos
    enablein name: addressed offset: 0 latency: 0
  end port
  port addressed
    comment
      "CPU address is valid"
    end comment
    transaction: data
    direction: input
    width: 1
```

```

    clock: clk polarity: pos
end port
port write
    comment
        "This is a write, not read"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
    enablein name: addressed offset: 0 latency: 0
end port
port data_in
    comment
        "Write data from CPU"
    end comment
    transaction: data
    direction: input
    width: 32
    clock: clk polarity: pos
    address port: addr
        physical: 31:0
        logical: 31:0
    enablein name: addressed offset: 0 latency: 0
end port
port data_out
    comment
        "Read data to CPU"
    end comment
    transaction: data
    direction: output
    width: 32
    clock: clk polarity: pos
    address port: addr
        physical: 31:0
        logical: 31:0
    enablein name: addressed offset: -1 latency: 0
end port
; Master side ports (to DDR2 RAM)
port output_enable
    comment
        "Address valid, read data"
    end comment
    transaction: data

```

A. Appendix

```
direction: output
width: 1
clock: clk polarity: pos
stallin name: stall offset: 0 latency: 4
end port
port byte_write_enable
comment
    "Address valid, write data bytes"
end comment
transaction: data
direction: output
width: 16
clock: clk polarity: pos
stallin name: stall offset: 0 latency: 4
end port
port ram_address
transaction: data
direction: output
width: 32
clock: clk polarity: pos
stallin name: stall offset: 0 latency: 4
end port
port write_data
transaction: data
direction: output
width: 128
clock: clk polarity: pos
address port: ram_address
    physical: 31:0
    logical: 31:0
stallin name: stall offset: 0 latency: 4
end port
port read_data
transaction: data
direction: input
width: 128
clock: clk polarity: pos
address port: ram_address
    physical: 31:0
    logical: 31:0
stallin name: stall offset: -1 latency: 4
end port
port stall
comment
```

```
        "FastLane+ is busy, wait"
    end comment
    transaction: data
    direction: input
    width: 1
    clock: clk polarity: pos
end port
end interface
end component
end comap
```

Listing A.3: CoMAP platform definition for Xilinx ML507

```
comap 1.0
; platform composition
platform ML507
  version: 1.0
  comment
    "Xilinx ML507"
  end comment
  configuration default
    version: 1.0
    comment
      "basic subset"
      "DDR2 controller is pulled in as resource"
    end comment
    component: PowerPC440
    component: Interrupt_Controller
    component: Ethernet
    component: UART
    component: HA require: 1.12
  end configuration
end platform
end comap
```


Index

- ABI, 21
- ACS, 13, 37, 47, 55, 117, 154, 177
- adaptive computer, 4, 13
- ADDR, 164
- ADDRESS, 42, 54, 56
- address, 82
- address range (gaussian distribution), 96
- ADL, 27, 28
- ADT, 85
- AE, 18
- AISLE, 23, 55
- ALU, 146
- AMBA, 22, 76
- API, 8, 54, 57, 60
- ARB, 30
- arbitration, 92, 123, 124
- architectural memory, 151, 158, 168, 175
- ATX, 197

- behavior, 127, 131
- bitonic sorter, 168
- Bitonic sorter module, 170
- block, 107
- boundary function, 72, 102
- BRAM, 21
- branch prediction, 154
- bridge, 107
- BTB, 167
- bus, 76
- bus bridge, 98

- cache, 168
- CachePort, 164, 169
- CAM, 32, 51, 159, 162, 165
- CAN, 15
- cancel, 149, 166

- CFG, 135
- clock, 82
- CMDFG, 134
- code correlation, 32
- coherency (memory), 152, 158
- CoMAP, 7, 173
- CoMAP-VPP, 64, 108
- commit, 149, 166, 175
- compiler, 133
- component, 98
- component (active), 104
- Comrade, 38, 51, 133
- configuration (platform), 104
- consumer, 87
- copy on write, 56
- CPU, 3, 4, 18, 30, 65, 115

- DDR, 4, 43, 46, 50, 52, 115, 197
- DDR2, 17
- demand-paging, 56
- dependency (anti), 149
- dependency (output), 150
- dependency (true), 149
- dependency prediction, 152
- dependency relation, 73
- destructive (memory), 151, 158
- disambiguation (memory), 151, 154
- DMA, 23, 53, 56, 59, 115
- DRAM, 15, 16
- DSP, 3, 98
- dynamic priority, 167
- dynamic profiling, 38

- EDA, 16, 97, 108
- EDK, 26, 44
- ELF, 56

- ESL, 2, 26
- execution model, 6, 146, 162
- execution schedule, 119
- expression, 69, 73

- FastLane+, 50, 52, 116
- FastPath, 49, 178
- FIFO, 51, 97, 119
- FIR, 98
- flip-flop, 3
- forwarding, 152
- FPGA, 162
- Front Side Bus, 18, 22
- FSB, 18, 22
- FSM, 29, 170
- function (C), 118, 130–132

- GAg, 190, 192
- GAg predictor, 166
- GAg predictor module, 170
- gaussian distribution, 95, 175
- general purpose processor, 3
- GID, 160, 162
- glibc, 58
- GPP, 13, 21, 37, 47, 50, 52, 57, 145, 154, 162, 166, 177, 194

- HA, 13, 47, 50, 52, 63, 115, 154, 158, 162, 177, 194
- HAMEM, 42, 50, 54, 60
- handshake, 85, 86, 88, 119, 124
- hardware accelerator, 3, 13, 135
- hazard (memory), 149, 155, 161
- HDL, 7, 9, 19, 26, 28, 63
- HDTV, 1
- HTM, 32

- IEC, 81
- ILP, 34, 38, 156
- inheritance, 75, 124
- intellectual property, 63, 74
- interface, 74, 81, 98, 101, 118, 123
- IP, 8, 63, 74
- IP core, 64, 98, 117, 118
- IR, 29, 134, 136, 141

- IRQ, 22, 47, 177

- latency, 88
- linear distribution, 97
- Linux, 43
- live variable, 40
- local distribution, 96, 175
- look-up table, 3, 59
- LOWLAT, 42, 47, 49
- LSQ, 30, 159, 162, 163
- LUT, 59, 169

- MAC, 115
- MARC, 51, 163, 166, 169, 187
- MARC II, 168, 192, 203
- MARTE, 27
- master, 92
- master mode, 40
- MCI, 115
- memory localization, 96
- memory management unit, 22
- ML310, 42, 46, 178
- ML507, 115
- MMU, 22, 53, 54, 59
- monolith, 49, 130
- MUXCY, 169

- N-bit predictor, 166
- non-cacheable, 54
- NRE, 2
- NUMA, 15

- OE, 164, 172
- offset, 88
- OP, 172
- OPB, 44
- OS, 13, 60, 177
- OSSCHED, 42, 52
- out-of-order execution, 154
- Output Prepare, 170

- PaCIFIC, 85, 117
- page table, 59
- parameter, 66, 69, 98, 101
- parameter (boundary function), 73

- parameter (dependent), 102
- parameter (free), 109
- partitioning (HW/SW), 38
- pattern-history table (predictor), 166
- PCB, 18
- PCI, 43
- PCI-Express, 16
- PCI-X, 26
- PDA, 1
- PE, 14
- PHT, 166, 175
- PHY, 115
- pipeline, 118, 146
- pipelining, 128
- platform management, 7, 44
- platform specification, 63
- PLB, 44, 46, 53, 115
- pointer, 56
- port, 79, 85
- prediction (access), 152, 164, 166, 168
- prediction (address), 154, 158
- prediction (control), 158, 163
- prediction (data), 154, 158
- prefetching, 88, 149, 163, 168
- primitive, 49, 130
- Priority module, 170
- producer, 87
- program order, 149
- programmed IO, 119
- PROTCODE, 42, 57
- PROTSYS, 42, 57, 60

- QDR, 18
- QoS, 97
- QPI, 22
- quality of service, 97

- RAM, 4, 50, 51, 53, 59, 74, 159, 162
- RAM16X1S, 168
- RAW, 149, 152, 157
- RCP, 13, 63
- RCU, 22
- RD, 13
- real-time OS, 4

- reconfigurable compute unit, 22
- Reconfigurable Computing Platform, 2, 116, 143, 146, 176
- reconfigurable SoC, 4
- register renaming, 162
- reorder buffer, 154
- repository, 64, 65, 110
- resource (platform), 65, 76
- retire (memory), 151
- RFID, 2
- rSoC, 17, 50, 63, 65, 115, 117, 196
- RTL, 7, 25
- RTOS, 13, 47

- scheduler (OS), 47, 52
- SCP, 19
- SDR, 51
- SDRAM, 4, 17, 115, 197
- semaphore, 48
- sequence, 85, 124, 130, 132
- sequence number, 157
- shared memory, 41
- SI, 81
- slave, 92
- SoC, 1, 63
- SPARC, 3
- speculation (address), 148
- speculation (control), 146, 163
- speculation (data), 148
- speculation (memory), 145
- speculation controller, 170
- SRAM, 4, 16, 17
- SRL16E, 168
- SSA, 134
- STALL, 164, 172
- store queue, 154
- streaming, 97, 119, 129, 133
- struct (C), 133
- SWPERF, 42, 55, 57

- technology, 101
- template, 65, 67, 71, 74–76, 101, 120, 124
- thread, 129
- TID, 160, 162

Index

TLB, 23, 59
traffic, 92, 94, 173
traffic class, 95
transaction (interface), 80, 87, 88
transaction (memory), 157
two-level predictor, 166, 173

UART, 44, 115
UML, 27
unique key (component), 101
unique key (platform), 107

V2P, 37, 179
Verilog HDL, 7, 9, 25, 26, 63, 69, 108, 169
VHDL, 7, 25, 63, 69, 108
VPP, 27, 108

WAR, 149, 157
WAW, 150, 151, 157
WE, 164, 172
WP, 172
write buffer, 154, 159
Write Prepare, 170
write update, 152

Academic Experience

10/1994–03/2001

Diploma in Computer Science at TU Braunschweig

04/2001–04/2003

Design Methodology Engineer at sci-worx GmbH, Braunschweig

05/2003–04/2005

Research Assistant at the Department of Integrated Circuit Design (E.I.S.),
TU Braunschweig

since 05/2005

Research Assistant at the Embedded Systems and Applications group (ESA),
TU Darmstadt

since 10/2008

Research Engineer at LOEWE Research Center AdRIA, TU Darmstadt